Delphi XE2 Foundations

Part 3: Packages, RTTI, Interoperability and Multithreading

Chris Rolliston

Delphi XE2 Foundations - Part 3

Updated 30 June 2012

Table of contents

General Introduction

Delphi is a complete software development environment for Windows, having its own language, libraries and 'RAD Studio' IDE. Now in its XE2 incarnation, Delphi continues to evolve. Most notably, XE2 supports writing applications for Mac OS X, an ability added in the context of numerous other extensions and improvements across recent releases.

Centred upon XE2, this book aims to provide a comprehensive introduction to the fundamentals of Delphi programming as they stand today. A contemporary focus means it has been written on the assumption you are actually using the newest version. Since a traditional strength of Delphi is how new releases rarely require old code to be rewritten, much of the text will apply to older versions too. However, the book does not explicitly highlight what will and what will not.

Conversely, a focus on fundamentals means a subject-matter of the Delphi language and wider runtime library (RTL). Topics such as the RAD Studio IDE and Visual Component Library (VCL — Delphi's longstanding graphical user interface framework) are therefore covered only incidentally. Instead, coverage spans from basic language syntax — the way Delphi code is structured and written in other words — to the RTL's support for writing multithreaded code, in which a program performs multiple operations at the same time.

If you are completely new to programming, this book is unlikely to be useful on its own. Nonetheless, it assumes little or no knowledge of Delphi specifically: taking an integrated approach, it tackles features that are new and not so new, basic and not so basic. In the process, it will describe in detail the nuts and bolts useful for almost any application you may come to write in Delphi.

About the Kindle edition of Delphi XE2 Foundations

Delphi XE2 Foundations is available in both printed and eBook versions. In eBook form, it is split into three parts; the part you are reading now is **part three**. All parts share the same ‘general introduction’, but beyond that, their content differs.

Chapter overview: part one

The first chapter of both the complete book and part one of the eBook set gives an account of the basics of the Delphi language. This details the structure of a program's source code, an overview of the typing system, and the syntax for core language elements. The later parts of the chapter also discuss the mechanics of error handling and the syntax for using pointers, a relatively advanced technique but one that still has its place if used appropriately.

Chapter two turns to consider simple types in more detail, covering numbers, enumerations and sets, along with how dates and times are handled in Delphi.

The third and fourth chapters look at classes and records, or in other words, Delphi's support for object-oriented programming (OOP). Chapter three considers the basics, before chapter four moves on to slightly more advanced topics such as metaclasses and operator overloading.

Chapter overview: part two

The first chapter of part two — chapter five of the complete book — considers string handling in Delphi. It begins with the semantics of the `string` type itself, before providing a reference for the RTL's string manipulation functionality, including Delphi's regular expressions (regex) support.

Chapter six discusses arrays, collections and custom enumerators, providing a reference for the first and second, and some worked-through examples of the third.

Chapters seven to nine look at I/O, focussing initially upon the basics of writing a console program in Delphi, before turning to the syntax for manipulating the file system (chapter seven), an in-depth discussion of streams (chapter eight), and Delphi's support for common storage formats (chapter nine).

Chapter overview: part three

Chapter ten — the first chapter of part three — introduces packages, a Delphi-specific form of DLLs (Windows) or dylibs (OS X) that provide a convenient way of modularising a monolithic executable. The bulk of this chapter works through a FireMonkey example that cross compiles between Windows and OS X.

Chapters eleven and twelve look at Delphi's support for dynamic typing and reflection — 'runtime-type information' (RTTI) in the language of Delphi itself. The RTTI section is aimed primarily as a reference, however example usage scenarios are discussed.

Chapter thirteen looks at how to interact with the native application programming interfaces (APIs). A particular focus is given to using the Delphi to Objective-C bridge for programming to OS X's 'Cocoa' API.

Chapter fourteen looks at the basics of writing and using custom dynamic libraries (DLLs on Windows, dylibs on OS X), focussing on knobbly issues like how to exchange string data in a language independent fashion. Examples are given of interacting with both Visual Basic for Applications (VBA) and C#/.NET clients.

Finally, chapter fifteen discusses multithreaded programming in Delphi, with a particular focus on providing a reference for the threading primitives provided by the RTL. This is then put to use in the final section of the book, which works through a 'futures' implementation based on a custom thread pool.

Trying out the code snippets

Unless otherwise indicated, the code snippets in this book will assume a console rather than GUI context. This is to focus upon the thing being demonstrated, avoiding the inevitable paraphernalia had by GUI-based demos.

To try any code snippet out, you will therefore need to create a console application in the IDE. If no other project is open, this can be done via the Tool Palette to the bottom right of the screen. Alternatively, you can use the main menu bar: select `File|New|Other...`, and choose `Console Application` from the `Delphi Projects` node. This will then generate a project file with contents looking like the following:

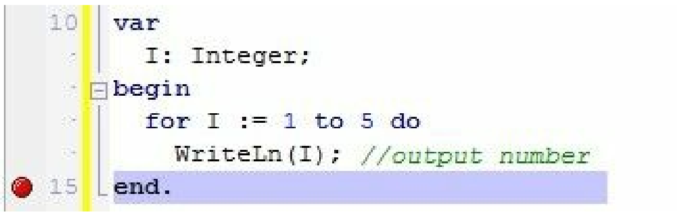
```
program Project1;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

Unless the context indicates otherwise, the `uses` to the final `end` should be overwritten with the code snippet.

By default, the console window will close when a debugged application finishes, preventing you from inspecting any output first. To avoid this, you can amend the final part of the code added to look like the following:

```
  //code...
  Write('Press ENTER to exit...',
  ReadLn;
end.
```

Alternatively, you can add a breakpoint at the end of the source code. Do this by clicking in the leftmost part of gutter beside the `end.`:



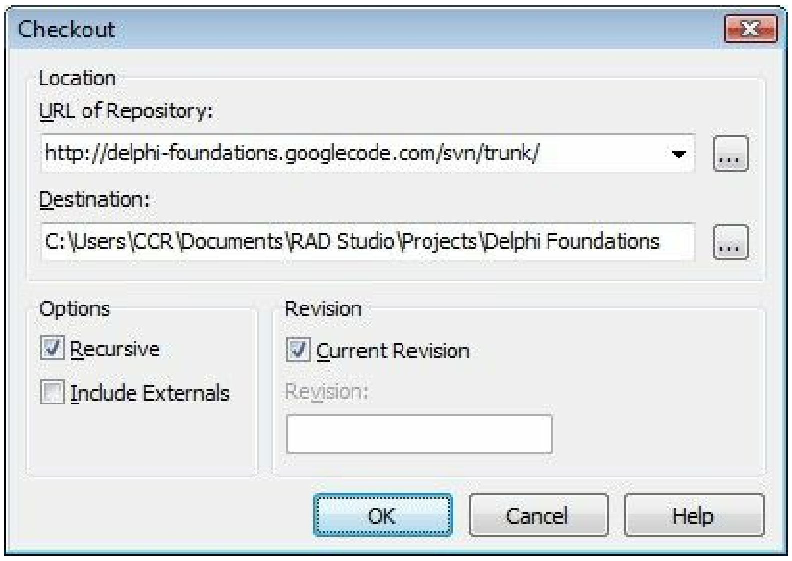
```
var
  I: Integer;
begin
  for I := 1 to 5 do
    WriteLn(I); //output number
end.
```

This will cause the IDE to regain control just before the program closes, allowing you to switch back to the console window to view the output. To allow the program to then close normally, switch back to the IDE and press either the ‘Run’ toolbar button or the F9 key, as you would have done to start the program in the first place.

Sample projects

Code for the more substantive examples in the book can be found in a Subversion repository hosted on Google Code. An easy way to download it is to use the RAD Studio IDE:

1. Choose `File|Open From Version Control...` from the main menu bar.
2. Enter `http://delphi-foundations.googlecode.com/svn/trunk/` for the URL, and a local folder of your choosing for the destination.


Checkout

Location
URL of Repository:
http://delphi-foundations.googlecode.com/svn/trunk/
Destination:
C:\Users\CCR\Documents\RAD Studio\Projects\Delphi Foundations

Options
[✓] Recursive
[ ] Include Externals

Revision
[✓] Current Revision
Revision:

3. On clicking OK, a new dialog should come up listing the sample projects as they get downloaded.
4. Once complete, a third dialog will display allowing you to pick a project or project group to open. It isn’t crucial to pick the right one though, since you can open any of the projects as normal afterwards.

Acknowledgements

I wish to thank the following, in no particular order: John Toelken for looking over chapters 1 and 3; Kenneth Cochran and Mark Walker for spotting some typos in chapters 4 and 8 respectively; Warren Postma for some concrete suggestions concerning chapter 13; Primož Gabrijelcic for technical editing of chapter 15; and Jasper Bongertz and James Chandler for assisting with tests on physical Kindle devices.

Chris Rolliston, 14 June 2012

10. Packages

Delphi ‘packages’ are a way to modularise monolithic executables, for example to break down a single EXE into a core application and several optional plugins. Unfortunately, packages compiled with one version of Delphi (where ‘version’ means, say, XE2, or 2010) will not work with executables compiled in another version, let alone written in another language. Nonetheless, if this limitation is acceptable for you, using them is the easiest way to implement a plugin architecture in Delphi.

A prime example of a Delphi program written in this way is the RAD Studio IDE. Here, different functionalities (the refactoring, the history viewer, the Subversion integration, etc.) are implemented as separate packages. This helps Embarcadero maintain different editions of the product, the Starter edition coming with a basic set of packages, the Professional edition adding a few more, the Enterprise edition adding some more again and so on. In the IDE, packages are also the technology behind the possibility to install (or uninstall) third party components without having to rebuild the IDE itself.

In this chapter, we will begin with a general overview of how to use packages, including details of relevant functionality provided by the RTL. In the abstract, there is not much to learn about them; however, putting packages to work can prove a somewhat ‘bitty’ experience in practice. In the second half of the chapter, we will therefore work through an extended example, writing a simple plugin-based FireMonkey application that cross compiles between Windows and OS X.

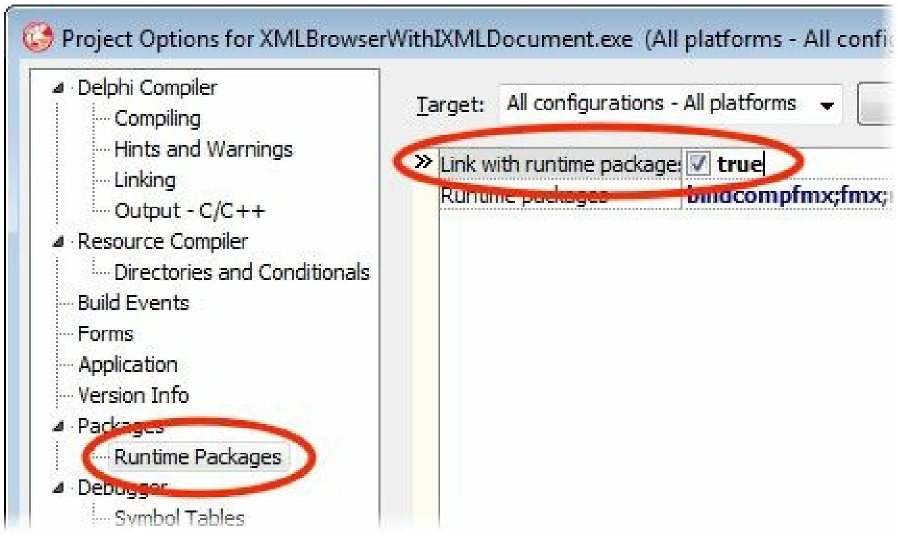
Package basics

Technically, Delphi packages are dynamic libraries (DLLs on Windows, dylibs on OS X) that expose Delphi-specific metadata. When an executable is built or 'linked' with a package, code that would have been compiled into the executable is instead left separately compiled in the package.

Originally, one of the reasons packages were added to Delphi was to reduce nominal executable size. This stems from how by default, every Delphi-authored executable will contain compiled code for those parts of Delphi's standard libraries that it uses. In the case of a suite of applications, this can mean the same code gets compiled into several files that all nonetheless have a relationship to each other. When you build with packages however — and in particular, link with the pre-built packages for the RTL, VCL and FireMonkey that Embarcadero provide with the product — this duplication will be avoided.

Linking with runtime packages

To see this in action, open up a project in the IDE and chose `Project|Options` from the main menu bar. In the dialog that results, select the 'Runtime packages' node on the left and tick the 'link with runtime packages' box on the right:



Link with runtime package: true
Runtime packages: bindcompfmx;fmx;

After a rebuild, the executable will now be smaller than before (perhaps much smaller), but at the cost of having required dependencies on one or more compiled packages or 'BPL files', so called because a compiled package usually has a `.bpl` extension on Windows and a `bpl` prefix on the Mac.

What dependencies arise exactly will depend on the units that are used across the application, together with the list of possible packages in the Project Options dialog: for each package listed, if the application uses at least one of its units, then the package is linked to. This linkage will be strong, so that if the operating system cannot find a linked BPL when your program is run, the program itself will fail to load. Nonetheless, linking to packages in this way allows the source code of the program code to remain unchanged, using the now packaged units as normal.

Even so, it is rare to build with any old package. This is because any changes in the `interface` section of one or more packaged units will require recompilation of not just the package itself, but any applications that have been built with it. In the case of the prebuilt packages for the RTL and VCL, this limitation is OK, since Embarcadero have a policy of product updates not breaking programs built with stock RTL and VCL packages when the current version (XE2, 2010, whatever) was first released. Otherwise, the need to recompile everything on a package interface change can be a deal-breaker.

In the case of packages you write yourself, an alternative to building with them is to dynamically load them at runtime instead. This loses some ease of use (you can no longer just use a packaged unit as normal), however increased robustness is acquired in return.

Creating your own packages

Since it is one of the standard project types, normally you create your own packages through the IDE. After doing this, the Project Manager will show a new node for the package, but not much else. This is normal; just add units to the project as you would usually do, and things will be fine.

Like a project for an application, a package project has a compiling project file. This has a DPK rather than a DPR extension though, and takes a slightly different form to a DPR:

```
package MyPackage;

{ Compiler options }

requires
  OtherPackage1, OtherPackage2;

contains
  Unit1, Unit2;

end.
```

In the IDE, you can open the DPK in the editor via `Project|View Source` from the main menu bar, or right-clicking on the package's node in the Project Manager and choosing `View Source`.

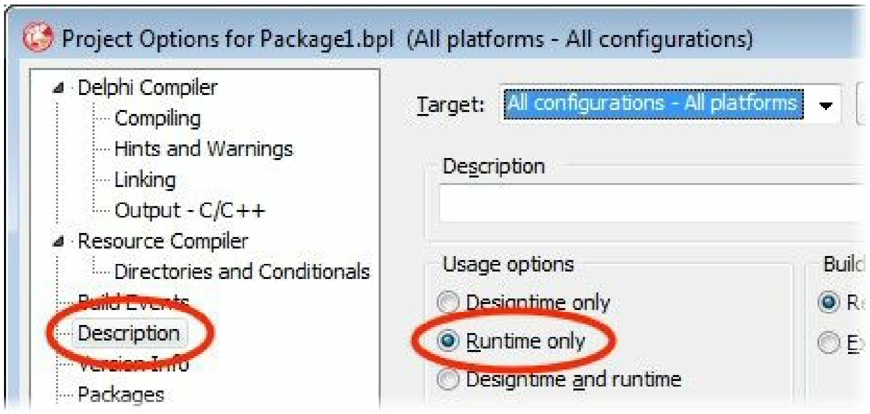
Unlike a DPR, a DPK has neither a `uses` clause nor `begin`/`end` block — instead, there are `requires` and `contains` clauses, and just an `end` to finish off. Conceptually, a package is a group of units. Thus, the `contains` clause lists the names of the units the package groups. The `requires` clause, in contrast, lists the names of any other packages your own one needs to link to. This will be the case when your package contains something that uses a unit grouped into some other package.

Typically, those other packages will be one or more of the packages that are shipped with Delphi itself, e.g. `rtl` (which contains the core RTL units) and `vcl` (which contains the core VCL units). The reason for this is that whenever a unit is used by both an application and a package loaded by the application, it cannot be contained in both. Thus, both application and package must be built (strong linked) to a second package that contains the shared unit. Consequently, if you have a custom package that implements a plugin form for a FireMonkey application, for example, both package and application should be set up to require the `rtl` and `fmx` packages.

Types of package

Packages come in two main types, 'design-time' and 'runtime'. Ultimately this is a licensing rather than a truly technical distinction, since only design-time packages are allowed to link against IDE internals.

Annoyingly, when you create a new package project in the IDE, it is tagged to be *both* a design-time *and* a runtime one. You should always change this setting to be one or the other — do so via the Description node of the Project Options dialog:



Usage options: Designtime only [✓ Runtime only] Designtime and runtime

From that same node you can set a textual description for the package too. This is only crucial for design-time packages however, since it defines the text to display in the IDE's Install Components dialog box.

Package files

When compiled, a package project produces a number of files: alongside the BPL, a DCP (Delphi Compiled Package) is outputted together with one DCU file for every contained unit. In order for an application to be built (strong linked) to a package, the compiler needs to find the DCP (the DCP isn't used otherwise). The DCU files, in contrast, are needed for a design-time package when an application using the components it contains statically links to them (i.e., is not built with runtime packages itself). In that case, the package's DCU files must be in a directory located on the IDE's library path. If they are not, confusing 'cannot find unit' compiler errors will result for components that nevertheless work fine in the form designer. If a runtime package is dynamically loaded at runtime however, only the BPL is needed.

Dynamically loading packages at runtime

To load a package in code, pass the file name of the BPL (preferably including its full path) to the `LoadPackage` function, as defined in `System.SysUtils`. If the package can't be loaded, an `EPackageError` exception is raised, otherwise all the `initialisation` sections (if any) of the package's units are run and a numeric handle to the package is returned. This handle is typed to `HMODULE` (sometimes you'll come across `HINST` instead, however they mean the same thing).

When a package is loaded in this way, the main application can no longer use its units as normal. Instead, you have two choices: either have the package explicitly say what types it exposes via exported functions, or have the main application poll the RTTI (runtime type information) system.

In the first case, a package is treated like an ordinary DLL or dylib, only without its author needing to worry about Delphi-specific types, calling conventions, and inter-module memory management. (We will be looking at writing DLLs and dylibs in some detail in a later chapter.) For example, the package could contain a unit that explicitly exposes a couple of form classes like this:

```
unit ExportDeclarations;

interface

implementation

uses
  FMX.Forms, SomeFormUnit, SomeOtherFormUnit;

type
  TFormClass = class of TForm;

function GetExportedFormClasses: TArray<TFormClass>;
begin
  Result := TArray<TFormClass>.Create(TSomeForm, TSomeOtherForm);
end;

exports
  GetExportedFormClasses
    {$IFDEF MACOS}name '_GetExportedFormClasses'{$ENDIF};
end.
```

After calling `LoadPackage`, the host application would then use the `GetProcAddress` function (part of the native API on Windows, part of the Delphi RTL on the Mac) to get a pointer to the `GetExportedFormClasses` routine:

```
uses
  {$IFDEF MSWINDOWS}Winapi.Windows,{$ENDIF}
  System.SysUtils, FMX.Forms;

type
  TFormClass = class of TForm;
  TGetExportedFormClassesFunc = function : TArray<TForm>;

var
  PackageHandle: HMODULE;
  Func: TGetExportedFormClassesFunc;
  ExportedFormClasses: TArray<TForm>;
begin
  PackageHandle := LoadPackage(PluginPath);
  Func := GetProcAddress(PackageHandle, 'GetExportedFormClasses');
  if @Func <> nil then
    ExportedFormClasses := Func;
```

A packaged form would then be instantiated as you would normally instantiate a form via a class reference:

```
var
  Form: TForm;
begin
  Form := ExportedFormClasses[0].Create(Application);
  Form.Show;
```

The second method of locating relevant types in a package is to poll the RTTI system. In that case, the host application will itself decide what types it is interested in. In the following example, it looks for FireMonkey HD form classes with 'Plugin' in their name:

```
uses
  System.SysUtils, System.Classes, System.Rtti, FMX.Forms;

type
  TFormClass = class of TForm;

function IsPluginFormType(AType: TRttiType;
  var AFormClass: TFormClass): Boolean;
var
```

```
  LClass: TClass;
begin
  if not (AType is TRttiInstanceType) then Exit(False);
  LClass := TRttiInstanceType(AType).MetaclassType;
  Result := LClass.InheritsFrom(TForm) and
    (Pos('Plugin', LClass.ClassName) > 0);
  if Result then
    AFormClass := TFormClass(LClass);
end;

function GetPluginFormClasses(
  PackageHandle: HMODULE): TArray<TFormClass>;
var
  Context: TRttiContext;
  FormCount: Integer;
  FormClass: TFormClass;
  LType: TRttiType;
  Package: TRttiPackage;
begin
  Result := nil;
  FormCount := 0;
  for Package in Context.GetPackages do
    if Package.Handle = PackageHandle then
    begin
      for LType in Package.GetTypes do
        if IsPluginFormType(LType, FormClass) then
        begin
          if Length(Result) = FormCount then
            SetLength(Result, FormCount + 8);
          Result[FormCount] := FormClass;
          Inc(FormCount);
        end;
      Break;
    end;
  SetLength(Result, FormCount);
end;
```

We will be looking at the RTTI system in much more detail in chapter 12.

Unloading dynamically loaded packages

Once a package has been loaded using `LoadPackage`, it can be unloaded once finished with by calling `UnloadPackage`. This routine is passed the numeric handle that `LoadPackage` returned; `UnloadPackage` then first calls the `finalization` sections (if any) of packaged units before actually unloading the BPL from memory.

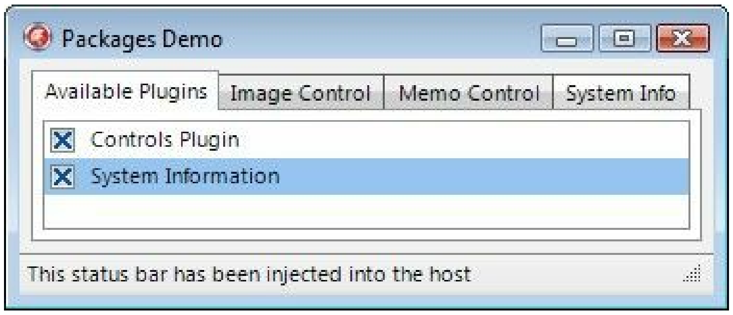
Prior to calling `UnloadPackage`, you must be careful to ensure there are no objects created from packaged types still alive. If there are, then access violations will quickly result due to the memory occupied by the objects no longer being valid. Ideally, objects would be kept track of as a matter of course, but if need be, you can use the `FindClassHInstance` or `FindHInstance` functions to help you find stray references:

```
procedure UnloadPackageAndItsForms(PackageHandle: HMODULE);
var
  Form: TForm;
  I: Integer;
begin
  for I := Screen.FormCount - 1 downto 0 do
  begin
    Form := Screen.Forms[I];
    if FindClassHInstance(Form.ClassType) = PackageHandle then
      Form.Free;
  end;
  UnloadPackage(PackageHandle);
end;
```

Packaged plugins: a FireMonkey example

To put some flesh onto the bones of all this, we will walk through an extended example of using packages in a plugin situation. The example will be using the FireMonkey framework, though it could be easily rewritten to use the VCL instead.

In it, the core application will be composed of a form with a tab strip. The first tab will show a list of available plugins, each implemented as a separate package and providing the content for one or more additional tabs. For illustrative purposes, we'll implement two plugins, one providing two tabs and the other a third:

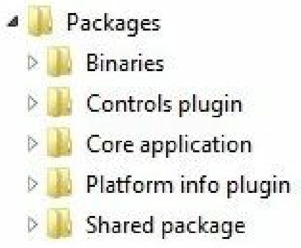


Both the main executable and individual plugins will live in the same folder, which on the Mac implies them all being inside the same 'bundle' (a native OS X application lies inside a special directory structure — the 'bundle' — that Finder treats as one thing by default). Nonetheless, executable and plugins will still be formally distinct in the sense the core application will run regardless of what plugins are installed.

In all, the application will be composed of four separate projects: a package containing code shared between executable and plugins; the executable project itself; and two plugin projects. When deployed, the stock `rtl` and `fmx` packages will also be required. In principle there could be any number of plugins (including none at all), but we will stick with just a couple.

Creating the project group

For ease of configuration, set up a directory structure like the following, so that each project gets its own sub-directory, with a further sub-directory defined for the compiled executable and BPL files:



Next, create the projects:

- Create a new package (`File|New|Package – Delphi`) and save it as `Shared.dpk` under the `Shared package` sub-directory.
- Via `Project|Add New Project...`, create a new FireMonkey HD application and save it as `PackagesDemo.dpr` under `Core application`.
- Use `Project|Add New Project...` twice more to create two further packages. Save them as `ControlsPlugin.dpk` and `SysInfoPlugin.dpk` under their respective directories.
- Finally, save the project group itself — the IDE should prompt you to do this anyhow if you do a `File|Save All`. Call the project group `Packages example.groupproj`, and save it in the root directory (i.e., one level up from each of the DPK and DPR files).

Configuring the output directories

The second step is to configure the projects' output directories. By default, a package is set up on the basis it should be globally available. While that is a good assumption when writing a package to be installed into the IDE, it is not so good in our case. To correct it, we need to change where the BPL and DCP files are outputted to for each package, before configuring internal search paths accordingly.

So, for each of the packages in turn, do the following:

- If it isn't already, ensure it is the 'active' project by double clicking on its node in the Project Manager to the right of the screen.
- Open up the Project Options dialog box (`Project|Options...`), and select the 'Delphi Compiler' node at the top of the tree view.
- From the combo box labelled 'Target' at the top, select 'All configurations - All platforms'.
- Change 'DCP output directory' to

  ```
  .\$(Platform)\$(Config)
  ```

  This should be the same value as the default 'Unit output directory' (leave that one alone).
- Change the 'package output directory' to

  ```
  ..\Binaries\$(Platform)\$(Config)
  ```
- For the two plugin projects, change 'Search path' to

  ```
  ..\Shared package\$(Platform)\$(Config)
  ```

  This is so the compiler can find the shared package's DCP file.
- Select the 'Description' node and change 'Usage options' to 'Runtime only'. For the two plugin projects, set a suitable package description here as well ('Controls Plugin' and 'System Information'). In general, setting a description string isn't necessary for a runtime package. However, since it makes sense for our plugins to have an associated 'friendly name' to display to the user, we might as well use the mechanism implemented for design-time packages.

For the application, bring up the Project Options dialog once more and set the following, ensuring 'All configurations - All platforms' is selected first as before:

- With the 'Delphi Compiler' node selected, change 'Search path' to

  ```
  ..\Shared package\$(Platform)\$(Config)
  ```

  As previously, this so the compiler can find our shared package's DCP file.
- Change the 'output directory' to

  ```
  ..\Binaries\$(Platform)\$(Config)
  ```
- In the 'Runtime Packages' sub-node, enable runtime packages by ticking the box, then add `Shared` to the runtime package list.

Implementing the shared package

Ideally, any package shared between core application and plugins should contain as little code as possible. This is because any changes in the `interface` section of a unit contained in the shared package will require recompiling every other project too.

In our case, the shared package will contain a single unit that defines a single interface type. This type will be implemented by any packaged form that wishes to define a tab in the host application; on loading a plugin, the host application in turn will cycle through the plugin's types looking for implementers of the interface.

To define the interface type, make sure the shared package is the 'active' project in the project group before adding a new unit to it. Save the unit as `PluginIntf.pas`, and change its contents to be thus:

```
unit PluginIntf;

interface

uses
  FMX.Types;

type
  IPluginFrame = interface
  ['{799E63ED-F990-41D2-BF44-80984D38E2A1}']
    function GetBaseControl: TControl;
  end;

implementation

end.
```

In use, the tab caption will be taken from the packaged form's `Caption` property, and the tab contents will be that of the specified ‘base control’.

In a VCL program, we could get away with making this interface method-less, or perhaps even avoid defining an interface completely and use a custom attribute instead (attributes will be discussed later in this chapter). This is because ‘frames’ exist in the VCL as a realised concept — a VCL ‘frame’ is just like a form, only always hosted inside something else. Since FireMonkey does not (yet) have a frames implementation, the tab source must be a form that nominates a base control, for example a client-aligned `TLayout`. This base control can then be reparented to the host application’s tab control immediately after the form is created.

The last thing to do with the shared package is to compile it (`Ctrl+F9`). This should output the BPL and all-important DCP to the places previously specified.

Writing the core program

When using dynamically loading packages, we need to ensure all packaged objects are freed before unloading the package itself. To cover this, we’ll write a small helper unit to keep track of created objects.

For this, make the host application the active project, before adding a new unit to it. Save the unit as `PluginManager.pas`, and change its `interface` section to look like this:

```
interface

uses
  System.SysUtils, System.Classes;

function LoadPlugin(const AFileName: string): HMODULE;
procedure RegisterPluginComponent(APluginHandle: HMODULE;
  AComponent: TComponent);
procedure UnloadPlugin(AHandle: HMODULE);
procedure UnloadAllPlugins;
```

`LoadPlugin` will call `LoadPackage` before creating an internal `TComponentList` for the package, `RegisterPluginComponent` will add a component to that list, `UnloadPlugin` will free the list (destroying the components previous added in the process) before calling `UnloadPackage`, and `UnloadAllPlugins` will unload any previously loaded plugin not explicitly unloaded before.

To put this into code, amend the unit’s `implementation` section to look like the following:

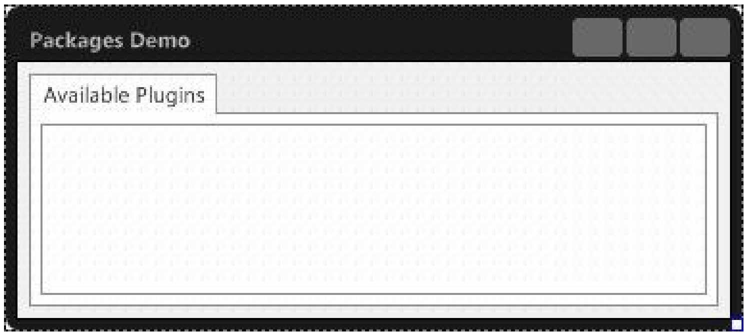
```
implementation

uses
  System.Contnrs, System.Generics.Collections;

var
  Dictionary: TDictionary<HMODULE,TComponentList>;

function LoadPlugin(const AFileName: string): HMODULE;
begin
  Result := LoadPackage(AFileName);
  Dictionary.Add(Result, TComponentList.Create(True));
end;

procedure RegisterPluginComponent(APluginHandle: HMODULE;
  AComponent: TComponent);
begin
  Dictionary[APluginHandle].Add(AComponent);
end;

procedure UnloadPlugin(AHandle: HMODULE);
begin
  Dictionary[AHandle].Free;
  Dictionary.Remove(AHandle);
  UnloadPackage(AHandle);
end;

procedure UnloadAllPlugins;
var
  Pair: TPair<HMODULE,TComponentList>;
begin
  for Pair in Dictionary.ToArray do
  begin
    Pair.Value.Free;
    UnloadPackage(Pair.Key);
    Dictionary.Remove(Pair.Key);
  end;
```

```
end;

initialization
  Dictionary := TDictionary<HMODULE,TComponentList>.Create;
finalization
  UnloadAllPlugins;
  Dictionary.Free;
end.
```

This code leverages the `TComponent` ownership pattern: when a root component like a form is freed, all the components it 'owns' (which for a form, will be all the controls and non-visual components placed on it at design time) will be freed too.

This helper unit implemented, we can now design the main form. Using the Object Inspector, first change the form's name to `frmMain`. Next, add a `TTabControl` to it; set the tab control's `Align` property to `alClient`, all elements of its `Margins` and `Paddings` properties (i.e., `Left`, `Top`, `Right` and `Bottom`) to 6, and its `Name` property to just `TabControl`.

Right click the tab control and select `Add Item`; change the `Text` property of the new tab to `Available Plugins`, then add to the tab a `TListBox`. After making sure the list box is actually parented to the tab rather than the form, set its `Align` property to `alClient`, its `DisableFocusEffect` and `ShowCheckboxes` properties to `True`, and its `Name` property to `lsbPlugins`. After all that, the form in the designer should look like this:



Now head to the form's source code (`F12`). Add to the top of the `implementation` section the following `uses` clause:

```
uses System.Rtti, PluginIntf, PluginManager;
```

Next, define a couple of helper functions, one to check for whether a given type exposed through the RTTI system is a suitable implementer of `IPluginFrame`, and the second to wrap the `GetPackageDescription` function. This wrapper will return `False` rather than an exception when passed the file name of an invalid package:

```
type
  TCustomFormClass = class of TCustomForm;

function IsPluginFrameClass(AType: TRttiType;
  var AFormClass: TCustomFormClass): Boolean;
var
  LClass: TClass;
begin
  if not (AType is TRttiInstanceType) then Exit(False);
  LClass := TRttiInstanceType(AType).MetaclassType;
  Result := LClass.InheritsFrom(TCustomForm) and
    Supports(LClass, IPluginFrame);
  if Result then
    AFormClass := TCustomFormClass(LClass);
end;

function TryGetPluginDescription(const AFileName: string;
  var ADescription: string): Boolean;
begin
  Result := True;
  try
    ADescription := GetPackageDescription(PChar(AFileName));
  except
    on EPackageError do
      Result := False;
  end;
end;
```

Go back to the Object Inspector, find the form's `OnCreate` event, and double click it to create a stub handler. In it we will insert code that searches for installed plugins. Do so like this:

```
procedure TfrmMain.FormCreate(Sender: TObject);
var
  Description: string;
  NewItem: TListBoxItem;
  Path: string;
  SearchRec: TSearchRec;
begin
  Path := ExtractFilePath(ParamStr(0));
  if FindFirst(Path + '*Plugin*', 0, SearchRec) <> 0 then Exit;
  try
    repeat
      if TryGetPluginDescription(Path + SearchRec.Name,
        Description) then
      begin
        NewItem := TListBoxItem.Create(lsbPlugins);
        NewItem.Text := Description;
        NewItem.TagString := Path + SearchRec.Name;
        lsbPlugins.AddObject(NewItem);
      end;
    until FindNext(SearchRec) <> 0;
  finally
    FindClose(SearchRec);
  end;
  lsbPlugins.Sorted := True;
end;
```

Here, plugins are found on the basis their BPLs (a) have been placed in the same directory as the main executable and (b) have the word 'Plugin' in their file name. For each one found, an item is added to the list box, with the BPL's full path stored in the item's `TagString` property.

Next, handle the list box's `OnChangeCheck` event like this:

```
procedure TfrmMain.lsbPluginsChangeCheck(Sender: TObject);
var
  Item: TListBoxItem;
begin
  Item := (Sender as TListBoxItem);
  if Item.IsChecked then
    Item.Tag := NativeInt(LoadPluginTabs(Item.TagString))
  else
  begin
    UnloadPlugin(HMODULE(Item.Tag));
    TabControl.Realign;
  end;
end;
```

In this code we use the list box item's `Tag` property to store the package handle when loaded. The casts are to be on the safe side, since while `Tag` is a *signed* pointer-sized integer, an `HMODULE` (what `LoadPackage` returns) is an *unsigned* pointer-sized integer.

The final bit of code for the main form is the `LoadPluginTabs` method called by the `OnChangeCheck` handler we have just defined. After declaring the method in the `private` or `strict private` section of the class, implement it like this:

```
function TfrmMain.LoadPluginTabs(const AFileName: string): HMODULE;
var
  Context: TRttiContext;
  Frame: TCustomForm;
  FrameClass: TCustomFormClass;
  LType: TRttiType;
  Package: TRttiPackage;
  Tab: TTabItem;
begin
  Result := LoadPlugin(AFileName);
  try
    for Package in Context.GetPackages do
      if Package.Handle = Result then
      begin
        for LType in Package.GetTypes do
          if IsPluginFrameClass(LType, FrameClass) then
          begin
            Tab := TTabItem.Create(TabControl);
            RegisterPluginComponent(Result, Tab);
            Frame := FrameClass.Create(Tab);
            Tab.Text := ' ' + Frame.Caption;
            (Frame as IPluginFrame).GetBaseControl.Parent := Tab;
            TabControl.AddObject(Tab);
            Tab.Width := Tab.Canvas.TextWidth(Tab.Text + 'w');
          end;
        Break;
      end;
  except
    UnloadPlugin(Result);
    raise;
  end;
end;
```

After loading the plugin, we cycle through the RTTI system's list of packages to find the one we just loaded. Once matched, we loop through the package's types looking for `IPluginFrame` implementers. For each one found, a new tab is created and associated with the plugin, and the tab's content set to the frame's declared base control. During all this, if and when an exception arises, the plugin is unloaded before the exception is re-raised. This is because we don't want any partially loaded plugins lying around.

Finally, compile the project. If you get an error saying the `PluginIntf` unit can't be found, either you haven't compiled the shared package yet, or the DCP paths weren't configured properly. Otherwise, the application should run, though without showing any plugins available since none have been written yet. Let's implement them now.

Implementing the 'controls' plugin

With `ControlsPlugin.dpk` now the active project, go into the DPK source and add `Shared` to the `requires` clause. After saving the project, add two FireMonkey HD forms to it. Set the `Caption` property of the first to `Image Control` and the second to `Memo Control`, before adding a `TMemo` to the first form, and a `TImageControl` to the second. Set the `Align` property of both controls to `alClient`, and add a picture of your choice to the image control — do this by double clicking inside the `Bitmap` property shown in the Object Inspector.

For each form in turn, switch to their code view (`F12`). Add `PluginIntf` to the `interface` section uses clause, and declare the form class to implement `IPluginFrame`. Then, actually implement `GetBaseControl` by returning either the memo or image control as appropriate. Here's what the code behind the completed memo form should look like (the image form will be similar):

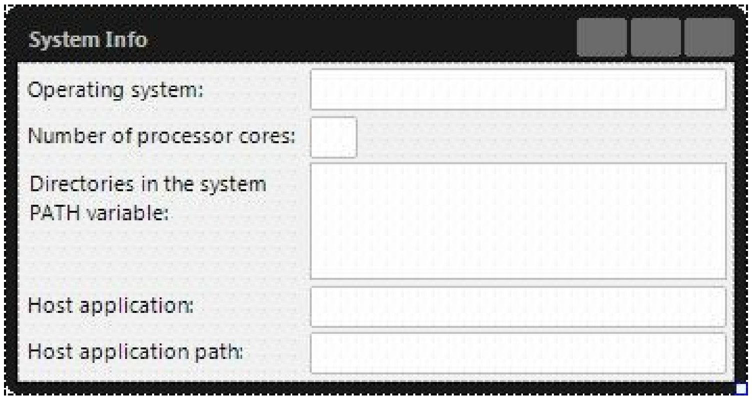
```
unit MemoFrame;

interface

uses
  System.SysUtils, System.Types, System.UITypes, System.Classes,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.Dialogs, FMX.Layouts,
  FMX.Memo, PluginIntf;

type
  TfrmMemo = class(TForm, IPluginFrame)
    Memo: TMemo;
  protected
    function GetBaseControl: TControl;
  end;

implementation

{$R *.fmx}

function TfrmMemo.GetBaseControl: TControl;
begin
  Result := Memo;
end;

end.
```

Compile the package, and rerun the main application (you won't need to recompile the host first, since that's the whole point of having a plugin architecture!); the newly-minted 'Controls Plugin' should now be listed, with two new tabs appearing when ticked (or crossed, depending on the visual style).

Implementing the 'system information' plugin

The second plugin will be in a similar vein as the first: as before, add `Shared` to the DPK's required clause, before adding a new FireMonkey HD form. To the form itself add a `TLayout` control this time, though client-align it as previously, and return it when implementing `GetBaseControl`.

Back in the form designer, add labels, edits and a memo control to the base layout so that the form looks like this:



Assuming the controls have been named appropriately, handle the form's `OnCreate` event as thus:

```
procedure TfrmSystemInfo.FormCreate(Sender: TObject);
begin
  edtOS.Text := TOSVersion.ToString;
  edtCPUCoreCount.Text := IntToStr(TThread.ProcessorCount);
  memPathEnvVar.Lines.Delimiter := PathSep;
  memPathEnvVar.Lines.DelimitedText := GetEnvironmentVariable('PATH');
  edtHostAppTitle.Text := Application.MainForm.Caption;
  edtHostAppPath.Text := GetModuleName(MainInstance);
end;
```

Inside a package, the `MainInstance` variable provides an `HMODULE` to the host application; passing an `HMODULE` to `GetModuleName` then returns the path to the module concerned. (To get the path to the package from within the package itself, use `HInstance` rather than `MainInstance`; inside an executable, both `MainInstance` and `HInstance` are still available, however they return the same thing.)

As hinted at by the assignment of `edtHostAppTitle.Text` here, the code inside a package can freely access internals of the host application so long as those internals are surfaced somehow. Simply reading off the main form's caption hardly shows this possibility off though. So, let's go further, and add a control to the host without the host having to do anything itself.

Injecting controls into the host form automatically

With dynamically loaded packages like our plugins, we can assume the main form of the host application will already be created by the time the package itself is loaded. Consequently, the package can simply inject what it likes in the `initialization` section of a contained unit, and clean up in a corresponding `finalization` section. Or at least, that's the basic principle. There is one small wrinkle though: in both FireMonkey and the VCL, child controls are automatically destroyed when their parents are destroyed, regardless of whether their parent is also their 'owner'. Because of this, the `finalization` section of a package should be careful not to call `Free` on what has become a 'stale' reference, something possible if the packaged control was injected into a form that was then freed before the package itself was unloaded.

To demonstrate the sort of code needed, let's amend the 'Controls Plugin' package to automatically add a status bar to the host. So, after making that package the active project again, and add two more units to it. Save the first as `InjectedControlsHelper.pas` and the second as `InjectedStatusBar.pas`. Then, put the following code into the first:

```
unit InjectedControlsHelper;

interface

uses
  System.Classes, FMX.Types;

type
  TInjectedControl<ControlType: TControl> = record
    class function Create: ControlType; static;
  end;

function InjectedControlsOwner: TComponent;

implementation

var
  FOwner: TComponent = nil;

function InjectedControlsOwner: TComponent;
begin
```

```
  Result := FOwner;
end;

class function TInjectedControl<ControlType>.Create: ControlType;
begin
  Result := TComponentClass(ControlType).Create(
    InjectedControlsOwner) as ControlType;
end;

initialization
  FOwner := TComponent.Create(nil);
finalization
  FOwner.Free;
end.
```

A bit like the `PluginManager` unit we implemented for the host application, this one leverages `TComponent`'s built-in memory management: instead of freeing an injected control directly, we have it owned by an internal object which is freed in its stead. If the parent form has already been freed in the meantime, taking the injected control with it, no access violation will result because the owner will have been notified of the fact internally.

Helper unit implemented, `InjectedStatusBar` (and, potentially, other control-injecting units) can be coded like this:

```
unit InjectedStatusBar;

interface

implementation

uses
  FMX.Types, FMX.Controls, FMX.Forms, InjectedControlsHelper;

var
  StatusBar: TStatusBar;
  LabelControl: TLabel;

initialization
  StatusBar := TInjectedControl<TStatusBar>.Create;
  StatusBar.Align := TAlignLayout.alBottom;
  LabelControl := TLabel.Create(StatusBar);
  LabelControl.Align := TAlignLayout.alClient;
  LabelControl.VertTextAlign := TTextAlign.taCenter;
  LabelControl.Text := ' This status bar has been injected ' +
    'into the host application''s main form by the plugin';
  StatusBar.AddObject(LabelControl);
  Application.MainForm.AddObject(StatusBar);
end.
```

Other than the slightly altered manner in which the status bar is created, the rest is as you would initialise a control in code regardless of packages being used or not.

If you now recompile the package and rerun the host application, activating the package should cause a status bar to appear along with the extra tabs, with no errors arising when the package is unloaded.

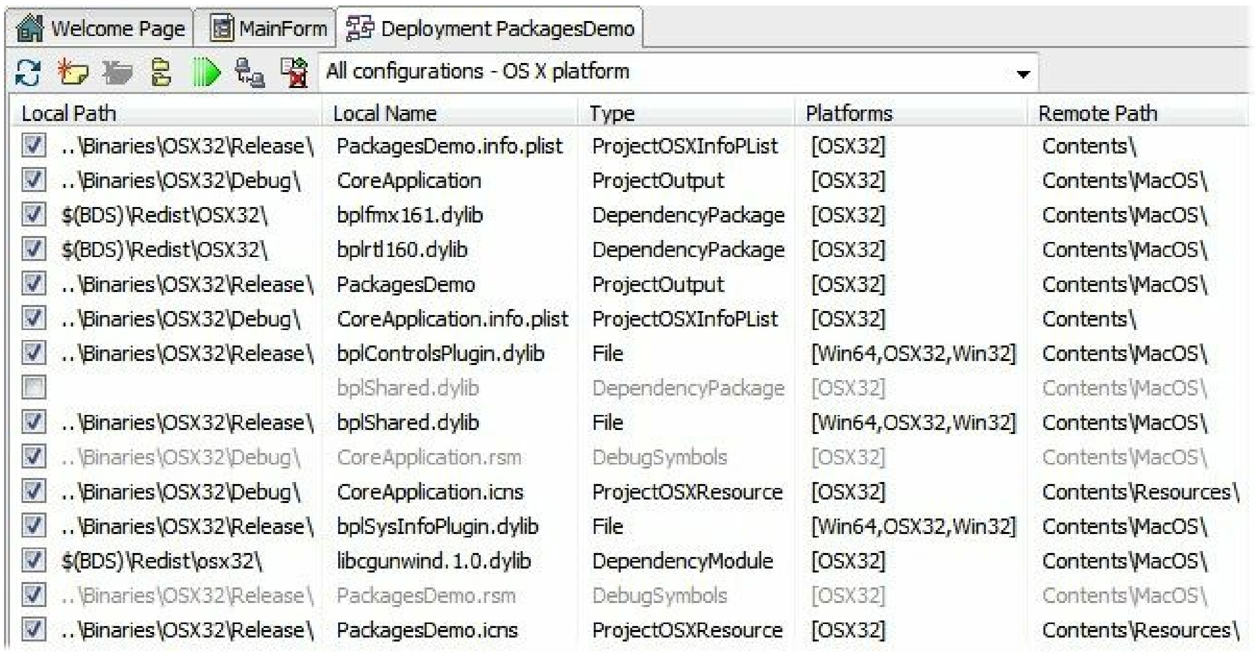
[Note: at my time of writing, there is a FireMonkey bug in which freeing a child control does not cause the parent to remove it visually until you have resized the parent.]

Setting things up when targeting OS X

If you are using the Professional or higher edition of Delphi or RAD Studio, changing each of the four projects' target platforms to Win64 and recompiling should lead the application and its plugins to work as before, only now as a 64 bit program. Simply changing the targets to OS X will not just work similarly however.

The reason is that the host application's deployment settings need to be configured to deploy the custom BPLs into the host application's 'bundle'. To do this, first compile each of the packages we created, using the 'Release' build configuration and OS X as the target platform. Next, set the host application as the active project, and select `Project|Deployment` from the main menu bar. This will cause a new 'Deployment' tab to be shown in the editor.

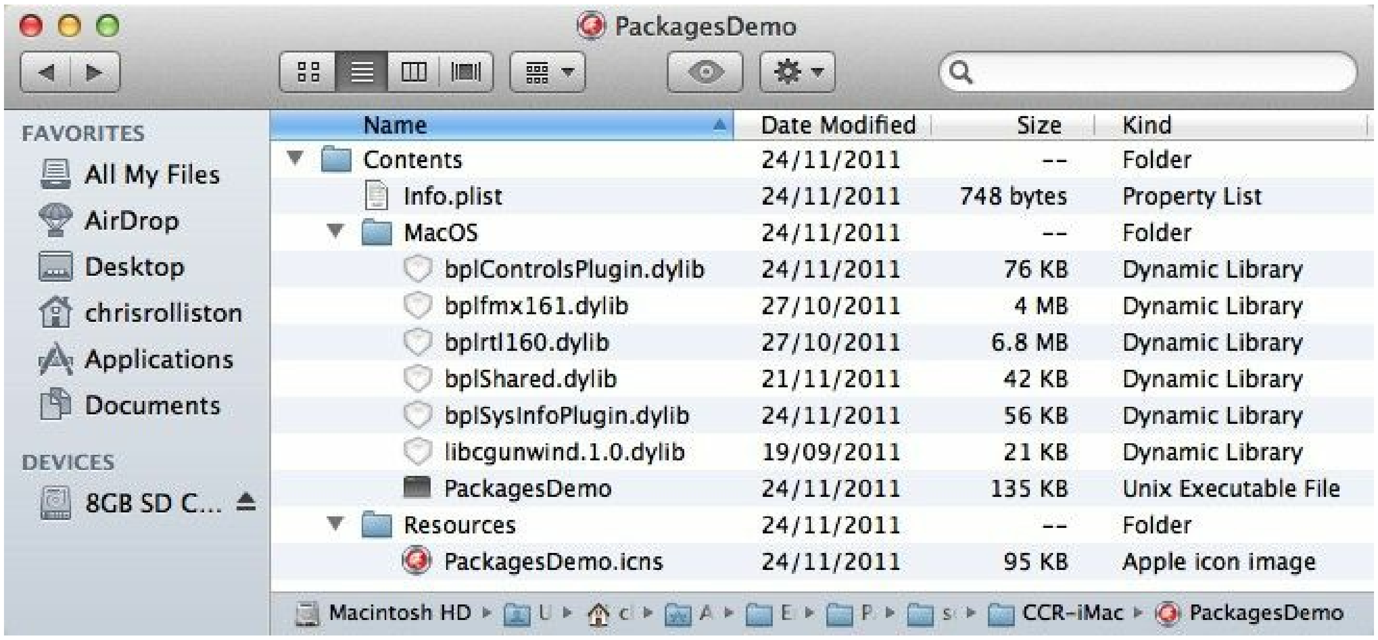
From the combo box at the top of it, select 'All Configurations — OS X platform'. Amongst the things listed, there will be a greyed-out entry for our Shared package. *Untick it*. Following that, click on the Add Files button (it's the second one from the left) and add our custom BPLs, as compiled for OS X (they should all be under `Binaries\OSX32\Release`).



	Local Path	Local Name	Type	Platforms	Remote Path
✓	..\Binaries\OSX32\Release\	PackagesDemo.info.plist	ProjectOSXInfoPList	[OSX32]	Contents\
✓	..\Binaries\OSX32\Debug\	CoreApplication	ProjectOutput	[OSX32]	Contents\MacOS\
✓	$(BDS)\Redist\OSX32\	bplfmx161.dylib	DependencyPackage	[OSX32]	Contents\MacOS\
✓	$(BDS)\Redist\OSX32\	bplrtl160.dylib	DependencyPackage	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Release\	PackagesDemo	ProjectOutput	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Debug\	CoreApplication.info.plist	ProjectOSXInfoPList	[OSX32]	Contents\
✓	..\Binaries\OSX32\Release\	bplControlsPlugin.dylib	File	[Win64,OSX32,Win32]	Contents\MacOS\
☐		bplShared.dylib	DependencyPackage	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Release\	bplShared.dylib	File	[Win64,OSX32,Win32]	Contents\MacOS\
✓	..\Binaries\OSX32\Debug\	CoreApplication.rsm	DebugSymbols	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Debug\	CoreApplication.icns	ProjectOSXResource	[OSX32]	Contents\Resources\
✓	..\Binaries\OSX32\Release\	bplSysInfoPlugin.dylib	File	[Win64,OSX32,Win32]	Contents\MacOS\
✓	$(BDS)\Redist\osx32\	libcgunwind.1.0.dylib	DependencyModule	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Release\	PackagesDemo.rsm	DebugSymbols	[OSX32]	Contents\MacOS\
✓	..\Binaries\OSX32\Release\	PackagesDemo.icns	ProjectOSXResource	[OSX32]	Contents\Resources\

After OKing everything, you should now be allowed to both compile and run the application.

If you do, an application bundle will be created on your Mac, by default somewhere under `~/Applications/Embarcadero/PAServer/scratch-dir/`. Using Finder, you can right click on the bundle and select 'Show Package Contents' to see what exactly was created. If you've made it this fair, something like the following should be shown (if you used the 'Debug' build configuration for the host application, the executable will be correspondingly larger, with a `.rsm` file alongside):



Name	Date Modified	Size	Kind
▼ Contents	24/11/2011	--	Folder
Info.plist	24/11/2011	748 bytes	Property List
▼ MacOS	24/11/2011	--	Folder
bplControlsPlugin.dylib	24/11/2011	76 KB	Dynamic Library
bplfmx161.dylib	27/10/2011	4 MB	Dynamic Library
bplrtl160.dylib	27/10/2011	6.8 MB	Dynamic Library
bplShared.dylib	21/11/2011	42 KB	Dynamic Library
bplSysInfoPlugin.dylib	24/11/2011	56 KB	Dynamic Library
libcgunwind.1.0.dylib	19/09/2011	21 KB	Dynamic Library
PackagesDemo	24/11/2011	135 KB	Unix Executable File
▼ Resources	24/11/2011	--	Folder
PackagesDemo.icns	24/11/2011	95 KB	Apple icon image

11. Dynamic typing and expressions

This chapter will have two parts. In the first, we will look at Delphi's support for dynamic typing in the shape of the `TValue` and `Variant` types. While this topic has some intrinsic interest, it will also be in preparation for what comes later, both in the present chapter and the next one. Following it, we will turn to the RTL's expression engine, a feature — new to XE2 — that allows you to compute simple (and sometimes not so simple) expressions entered by the user at runtime.

Dynamic typing in Delphi

At its core, Delphi is a strongly typed language, and that is how you should normally treat it: every variable, property, parameter or function result has a type, and only one type. Consequently, if you need to store or return a whole number, use `Integer`, and if you need to store a piece of text, use `string`, and so on.

Nonetheless, within the static type system sits types whose instances hold values that can be different sorts of thing at different times. These special types are `Variant` (along with its slightly more restricted sibling, `OleVariant`) and `TValue`. Use either of these, and the same variable can be assigned first a string, then a number, and so forth.

While `Variant` and `TValue` have overlapping concerns, their purposes are not identical however: `TValue` is a building block of Delphi's runtime type information (RTTI) system, and to that effect, can hold practically anything. Nonetheless, it does not support auto-conversions, where (for example) the string '123' can be automatically converted to the integer 123. Variants, in contrast, can do this. On the other hand, compared to `TValue`, variants are much more limited about the sorts of normal type they can hold. This stems from how their primary purpose is to support OLE Automation programming, in which case the full range of native Delphi types is not relevant.

TValue

`TValue` is defined by the `System.Rtti` unit, and while it plays an important role in Delphi's extended RTTI support, it can be used independently of that.

Its purpose is to serve as a general 'boxing' type, to borrow the .NET parlance: a variable typed to `TValue` serves as a generic 'box' in which you put in an instance of any other type, to be then 'unboxed' in a type safe manner later:

```
uses
  System.Rtti;

var
  Box: TValue;
  IntValue: Integer;
  StringValue: string;
begin
  Box := 42;
  IntValue := Box.AsType<Integer>;   //OK
  WriteLn(IntValue);
  StringValue := Box.AsType<string>; //runtime exception
end.
```

Having assigned (boxed) an instance of one type, there is nothing to stop you assigning an instance of a different type. In such a case, the type of the original value will just be forgotten about, or to use the boxing metaphor, the old contents will be thrown out before the new item is put in:

```
var
  Box: TValue;
  StringValue: string;
begin
  Box := 42;   //assign an integer
  Box := '42'; //assign a string: type of old value forgotten
  StringValue := Box.AsType<string>; //OK
end.
```

Assigning and retrieving data from a TValue

In the two examples so far, items have been boxed with simple assignments. This works for the more common types — floating point numbers, Booleans, classes, metaclasses and interfaces, as well as strings and integers. For any other type, use the `From<T>` static function:

```
uses
  System.Rtti;

type
  TMyEnum = (Zero, One, Two, Three);

var
  Box: TValue;
  EnumValue: TMyEnum;
begin
  EnumValue := One;
  Box := TValue.From<TMyEnum>(EnumValue);
  EnumValue := Two;
  WriteLn(Ord(EnumValue)); //2
  EnumValue := Box.AsType<TMyEnum>;
  WriteLn(Ord(EnumValue)); //1
end.
```

This example explicitly tells `From` which type to get (or more exactly, which version of `From` to instantiate). Usually, you won't need to do this due to generics' built-in type inference for simple cases like this:

```
Box := TValue.From(EnumValue);
```

A lower-level alternative to `From` is `Make`. This takes a pointer to the data to be boxed together with a pointer to the type information record for the data's type, which you get via the `TypeInfo` standard function. `Make` then returns a new `TValue` instance:

```
TValue.Make(@EnumValue, TypeInfo(TMyEnum), Box);
```

To unbox a value, you generally use the `AsType<T>` method as previously shown. For the types that support implicit assignment, there are also some dedicated methods if you wish to save a few keystrokes:

```
function AsBoolean: Boolean;
function IsClass: Boolean;
function AsCurrency: Currency;
function AsExtended: Extended; //i.e., as a floating point number
function AsInteger: Integer;
function AsInt64: Int64;
function AsInterface: IInterface;
function AsObject: TObject;
function AsString: string;
function AsVariant: Variant;
```

Where `AsInteger` has the same effect as `AsType<Integer>`, so `AsObject` has the same effect as `AsType<Integer>` and so on.

Similar to how `Make` is a lower level version of `From<T>`, the `AsType<T>` function has some lower-level peers too:

```
property DataSize: Integer read GetDataSize;
procedure ExtractRawData(ABuffer: Pointer);
procedure ExtractRawDataNoCopy(ABuffer: Pointer);
function GetReferenceToRawData: Pointer;
```

In the case of a reference type, `GetReferenceToRawData` will still return a pointer to the instance, even though the instance itself will be an implicit pointer already. If the `TValue` contains a string, for example, then you will need a cast to `PString` and dereference the pointer to read it as a `string`:

```
var
  Value: TValue;
  S: string;
begin
  Value := 'I love pointers';
  S := PString(Value.GetReferenceToRawData)^;
  WriteLn(S); //output: I love pointers
end.
```

Converting between types

As such, `TValue` is not designed for automatically converting between types. The general rule is that if a given conversion isn't supported implicitly in normal Delphi code, it isn't supported by `TValue`. Nonetheless, there are a couple of minor exceptions:

- `AsOrdinal` allows you to get the underlying integral value when the `TValue` holds an instance of an ordinal type, e.g. an enumeration.

- `ToString` returns a string representation of whatever is held:

```
const
  Arr: array[1..2] of string = ('First', 'Second');
var
  DynArr: TArray<Integer>;
  Metaclass: TClass;
  Obj: TEncoding;
  Rec: TSearchRec;
  S: string;
begin
  S := TValue.From(999).ToString;    //999
  S := TValue.From('test').ToString; //test
  S := TValue.From(nil).ToString;    //(empty)
  S := TValue.From(Arr).ToString;    //(array)
  DynArr := TArray<Integer>.Create(1, 2);
  S := TValue.From(DynArr).ToString; //(dynamic array)
  S := TValue.From(Rec).ToString;    //(record)
  Obj := TEncoding.UTF8;
  S := TValue.From(Obj).ToString;    //(TUTF8Encoding @ 01EC29A0)
  Metaclass := TStringBuilder;
  S := TValue.From(Metaclass).ToString;
    //(class 'TStringBuilder' @ 004180D0)
```

In the class and class reference cases, the figure after the @ sign is the memory address of the object and class reference respectively.

Testing for the type

There a few ways to test for the data type currently held in a `TValue` instance. For a small selection of types and sorts-of-type, you can use dedicated `IsXXX` functions:

```
function IsArray: Boolean;
function IsClass: Boolean;
function IsInstanceOf(AClass: TClass): Boolean;
function IsObject: Boolean;
function IsOrdinal: Boolean;
```

When `IsOrdinal` returns `True`, then `AsOrdinal` will work without raising an exception. To test for any other type, use the `IsType<T>` generic method:

```
if Box.IsType<TMyEnum> then //do something
```

This is also matched by a `TryAsType<T>` function:

```
var
  Str: string;
begin
  if Box.TryAsType<string>(Str) then ShowMessage(Str);
```

Lastly, to test for whether the `TValue` has been assigned in the first place, call the `IsEmpty` method. This will also return `True` if the instance has been assigned a `nil` value, either directly or from a variable assigned `nil`:

```
uses
  System.Rtti;

type
  TMyObject = class
  end;

var
  Box: TValue;
  Obj: TMyObject;
begin
  WriteLn(Box.IsEmpty); //output: TRUE
  Box := 123;
  WriteLn(Box.IsEmpty); //output: FALSE
  Obj := nil;
  Box := Obj;
  WriteLn(Box.IsEmpty); //output: TRUE
end.
```

Arrays and TValue

`TValue` exposes special helper methods for arrays to allow manipulating individual elements without unboxing first:

```
function GetArrayLength: Integer;
function GetArrayElement(Index: Integer): TValue;
procedure SetArrayElement(Index: Integer; const AValue: TValue);
```

These work with both static and dynamic arrays. When a static array uses a funky indexer, it is treated in the same way it would be when passed to a routine that has an ‘open array’ parameter, i.e. the actual indexer is transformed in a zero-indexed integral one:

```
uses
  System.Rtti;

var
  Box: TValue;
  Arr: array['a'..'b'] of string;
  I: Integer;
  S: string;
begin
  Arr['a'] := 'Hello';
  Arr['b'] := 'Goodbye';
  Box := TValue.From(Arr);
  WriteLn('Length: ', Box.GetArrayLength);
  for I := 0 to Box.GetArrayLength - 1 do
    WriteLn(Box.GetArrayElement(I).ToString);
end.
```

In general, `TValue` aims to mimic the semantics of the normal types it wraps. Because of this, assigning a dynamic array will only copy a reference to the original data, just like assigning one dynamic array to another would merely copy a reference to the first’s data to the second. Calling `SetArrayElement`, then, will change the original array:

```
uses
  System.Rtti;

var
  Box: TValue;
  Arr: array of string;
  S: string;
begin
  SetLength(Arr, 2);
  Arr[0] := 'Hello';
  Arr[1] := 'Goodbye';
  Box := TValue.From(Arr);
  Box.SetArrayElement(0, 'Hiya');
  WriteLn(Arr[0]); //output: Hiya
end.
```

Variants

Like a `TValue`, a variant is a sort of container that can hold values of lots of different types. Unlike `TValue`, it can auto-convert between sub-types:

```
var
  V: Variant;
begin
  V := 'This is a string';
  WriteLn(V); //This is a string
  V := 42;
  WriteLn(V); //42
  V := 'This is again a string ' + V;
  WriteLn(V); //This is again a string 42
end.
```

As this example shows, auto-conversions can be performed completely implicitly. In practice, using explicit casts is a good policy though, since leaving matters to the RTL can lead to undesirable, or at least ambiguous results. For example, what value might the following code output?

```
var
  V1, V2, V3: Variant;
  IntValue: Integer;
begin
  V1 := '99';
  V2 := '900';
  IntValue := V1 + V2 + 1;
  WriteLn(IntValue);
end.
```

If you guessed 1000, you guessed wrong: evaluating in a left-to-right fashion, `V1 + V2` concatenates two strings, producing `'99900'`, which is then auto-converted to an integer before being added to `1`, producing `99901`.

Variant sub-types

By default, the following are assignable to and from variants: integers, real numbers, `TDateTime` values, strings, interface references, Booleans and (in a slightly indirect way) dynamic arrays. The most noticeable absentee is classes. If you really must put a class instance into a variant, you have to typecast to and from `NativeInt`:

```
function PackObjectIntoVariant(Obj: TObject): Variant;
begin
  Result := NativeInt(Obj);
end;

function UnpackVariantFromVariant(const V: Variant): TObject;
begin
  Result := NativeInt(V);
end;
```

An unassigned variant has the special value `Unassigned` (*not* `nil`); to test for an unassigned variant, call the `VarIsEmpty` function. Both `Unassigned` and `IsEmpty` are defined in `System.Variants` — while variants are built into the language, most variant information and manipulation functions are in that unit. Another special variant value is `Null`, which is typically used in high level database programming — you can think of it meaning ‘not applicable’ as distinct from ‘not yet assigned’. Use the `VarIsNull` function to check for it:

```
var
  V: Variant;
begin
  WriteLn(VarIsEmpty(V)); //output: TRUE
  WriteLn(VarIsNull(V));  //output: FALSE
  V := Null;
  WriteLn(VarIsEmpty(V)); //output: FALSE
  WriteLn(VarIsNull(V));  //output: TRUE
```

To find what sub-type a variant currently has, call the `VarType` function. This returns a number that can be matched to one of the `varxxx` constants defined in the `System` unit:

```
varEmpty    = $0000;
varNull     = $0001;
varSmallint = $0002; //Int16
varInteger  = $0003; //Int32
varSingle   = $0004;
varDouble   = $0005;
varCurrency = $0006;
```

```
varDate     = $0007;
varOleStr   = $0008; //WideString
varDispatch = $0009; //IDispatch
varError    = $000A; //COM specific
varBoolean  = $000B; //technically WordBool not Boolean
varVariant  = $000C;
varUnknown  = $000D; //any interface other than IDispatch
varShortInt = $0010; //Int8
varByte     = $0011; //UInt8
varWord     = $0012; //UInt16
varLongWord = $0013; //UInt32
varInt64    = $0014;
varUInt64   = $0015;
varString   = $0100; //AnsiString (= string in old Delphis)
varUString  = $0102; //UnicodeString (= string in modern Delphis)
```

Also defined in the `System` unit are a series of similar-looking `vtxxx` constants. Be careful not to confuse them with the `varxxx` ones though, since they relate to something else.

Nulls

A unique property of `Null` is that it 'propagates' through an expression. As a result, if you perform an operation using two variants to produce a third, and one of the source variants is `Null`, `Null` will be returned:

```
uses
  System.Variants;

var
  V1, V2, V3: Variant;
begin
  V1 := 'First part';
  V2 := Null;
  V3 := V1 + V2;
  WriteLn(VarIsNull(V3)); //TRUE
end.
```

When a variant contains `Null`, assigning to an ordinary type will cause a runtime exception. Since this can be annoying in the case of strings especially, the `VarToStr` and `VarToStrDef` functions are defined. When passed a null variant, these return either an empty string or (in the case of `VarToStrDef`) a default string of your choice:

```
var
  V: Variant;
  S: string;
begin
  V := Null;
  S := 'The value is "' + VarToStr(V) + '"';
  S := 'The value is ' + VarToStrDef(V, '(empty)');
```

The precise behaviour of null values when entered into an expression is controlled by four global variables in `System.Variants`:

```
type
  TNullCompareRule = (ncrError, ncrStrict, ncrLoose);
  TBooleanToStringRule = (bsrAsIs, bsrLower, bsrUpper);

var
  NullEqualityRule: TNullCompareRule = ncrLoose;
  NullMagnitudeRule: TNullCompareRule = ncrLoose;
  NullStrictConvert: Boolean = True;
  NullAsStringValue: string = '';
```

When `NullEqualityRule` is `ncrLoose` (the default), testing for either equality or inequality against `Null` (i.e., `V = Null, V <> Null`) always produces a Boolean result. In contrast, `ncrError` will raise an exception, and `ncrStrict` will always return `False`.

Always returning `False` even when comparing two nulls may sound strange, and indeed, the apparent strangeness is why it isn't the Delphi default. However, it is consistent with the idea that nulls propagate. It is also the normal behaviour of Visual Basic for Applications, as used by (for example) Microsoft Access:

```
Sub VBATest()
  Dim V As Variant
  V = Null
  If V = Null Then
    MsgBox "Yep, set to Null"
  Else
    MsgBox "Er, not set to Null...?"
  End If
End Sub
```

Try this code out in any VBA environment, and a message box saying 'Er, not set to Null...?' will be shown. In contrast, the equivalent Delphi code wouldn't, so long as `NullEqualityRule` hasn't been changed:

```
uses System.Variants, FMX.Dialogs;

procedure Test;
var
  V: Variant;
begin
  V := Null;
  if V = Null then
    ShowMessage('Yep, set to Null')
  else
    ShowMessage('Er, not set to Null...?');
end;
```

As `NullEqualityRule` controls testing nulls for equality, so `NullMagnitudeRule` controls what happens when nulls appear in greater-than (>) or less-than (<) expressions: use `ncrLoose` (the default), and null variants are considered equivalent to unassigned ones.

`NullStrictConvert` controls what happens when you attempt to assign a `Null` variant to a normal type. In this case Delphi's default is the same as VBA's, which is to raise an exception:

```
var
  I: Integer;
begin
  I := Null; //runtime exception
```

Set `NullStrictConvert` to `False`, and nulls will once be treated like unassigneds — when assigning to an integer, for example, this will mean the number gets set to 0.

Lastly, `NullAsStringValue` defines the value `VarToStr` will assign in the case of a null source, assuming you haven't used the overload of `VarToStr` that explicitly specifies this. It also controls the value assigned when `NullStrictConvert` is `False` and `Null` to `string` assignment is made directly:

```
uses System.Variants;

var
  S: string;
begin
  NullStrictConvert := False;
  NullAsStringValue := '<null>';
  S := Null;
  WriteLn(S); //output: <null>
end.
```

Variants and arrays

While arrays can be boxed into a `TValue` like instances of any other type, 'variant arrays' are a distinct variant sub-type. You can create a variant array either explicitly via `VarArrayCreate`, or implicitly by assigning a dynamic array:

```
var
  V1, V2: Variant;
begin
  V1 := VarArrayCreate([1, 2], varInteger);
  V2 := TArray<Integer>.Create(98, 99, 100);
```

Only certain element types are supported, specifically numbers (e.g. `varInt64`, `varDouble`), date/times (`varDate`), 'OLE' strings (`varOleStr`), interfaces (`varUnknown`, `varDispatch`) and booleans. The notable one missing here is `varUString`, for native Delphi strings. However, the RTL will perform the necessary conversions implicitly if you assign a dynamic string array to a variant — an exception is only actually raised if `VarArrayCreate` is called directly and `varUString` passed:

```
var
  V: Variant;
begin
  V := TArray<string>.Create('First', 'Second'); //OK
  V := VarArrayCreate([0, 1], varUString);       //exception
```

The string case excepting, assigning a dynamic array to a variant is just a shortcut for calling `VarArrayCreate` and copying over the data manually. This produces different behaviour to assigning a dynamic array to a `TValue`, since in that case, just a reference to the array is copied, not the actual data:

```
uses System.Rtti, System.Variants;

var
  DynArray: TArray<Integer>;
  V: Variant;
  Value: TValue;
begin
  DynArray := TArray<Integer>.Create(12, 24);
  //assign to variant
  V := DynArray;
  V[0] := 42;
  WriteLn(DynArray[0]); //24, since array data was copied
  //assign to TValue
  Value := TValue.From(DynArray);
  Value.SetArrayElement(0, 42);
  WriteLn(DynArray[0]); //42, since array data was referenced
end.
```

Variant array routines

`VarArrayCreate` has the following signature:

```
function VarArrayCreate(const Bounds: array of Integer;
  AVarType: TVarType): Variant;
```

`AVarType` must be one of the `varXXX` constants we met earlier; `Bounds` specifies one or more pairs of lower and upper bounds. Pass four numbers, and you create a two dimensional array, pass six and you create a three dimensional one and so on:

```
var
  V: Variant;
begin
  V := VarArrayCreate([1, 2, 1, 3], varOleStr);
  V[1, 1] := 'First element of first dimension';
  V[1, 2] := 'First element of second dimension';
  V[1, 3] := 'First element of third dimension';
```

When creating a variant array from a dynamic array, the bounds match the source array, i.e. a lower bound of zero and an upper bound of the length minus one.

Aside from calling `VarArrayCreate` directly and assigning a dynamic array, a third way to create a variant array is to call `VarArrayOf`. This returns an array of variants:

```
function VarArrayOf(const Values: array of Variant): Variant;
```

The following uses this function to create a variant array of variants that contains a string, an integer and a floating point number:

```
var
  V: Variant;
begin
  V := VarArrayOf(['foo', 45, 9.842]);
```

To test whether a given variant contains an array, call `VarIsArray`. If it does, get and set individual elements using normal array indexing:

```
V1[1] := 356;
if V1[2] = 42 then //do something
```

If you wish to retrieve the dimensions of a variant array, dedicated functions must be used:

```
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
```

After a variant array has been allocated, you can change its length by calling `VarArrayReDim` (in the case of a multidimensional variant array, this will change the length of the rightmost dimension). This function is passed not the new length value directly, but the new high bound — in a variant array with bounds of 0 to 4, this means passing 5 if you wish to increase its size to 6 elements long:

```
var
  V: Variant;
begin
  V := VarArrayCreate([0, 4], varOleStr);
  //...
  VarArrayRedim(V, 5);
  V[5] := 'new value';
```

Accessing variant array data directly

Outside of normal array indexing, the RTL also enables low-level access to a variant array's data with the `VarArrayLock` and `VarArrayUnlock` functions:

```
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
```

`VarArrayLock` returns a pointer to the first element of the array; when successful, it should be paired with a call to `VarArrayUnlock`. The following function uses `VarArrayLock` to efficiently copy the data from a stream into a new variant byte array:

```
uses
  System.Classes, System.Variants;

function VarArrayCreateFromStream(Stream: TStream): Variant;
var
  DataPtr: PByte;
  Size: Integer;
begin
  Stream.Position := 0;
  Size := Stream.Size;
  Result := VarArrayCreate([0, Size - 1], varByte);
  DataPtr := VarArrayLock(Result);
  try
    Stream.ReadBuffer(DataPtr^, Size);
  finally
    VarArrayUnlock(Result);
  end;
end;
```

Historically, variant byte arrays were a popular way to marshal records between Visual Basic clients and COM servers written in more capable languages like Delphi.

Variants and COM

On Windows, Delphi's variant support wraps and extends the COM/OLE variant type: variants in Delphi, in other words, are essentially the same things as variants in Visual Basic for Applications, albeit without the special status of being the default data type. Consequently, the `varXXX` constants directly map to the API's `VT_XXX` ones, and variant arrays directly map to so-called 'safe arrays' in COM — indeed, you can get a pointer to the underlying safe array structure of a variant array via the `VarArrayAsPSafeArray` function.

In this way, variants form a core part of Delphi's support for OLE Automation, where variants form a standard parameter and return type. On Windows, Delphi also implements VB-style COM scripting on variants too:

```
uses System.Win.ComObj, System.Variants;

var
  WordApp, NewDoc: Variant;
begin
  WordApp := CreateOleObject('Word.Application');
  NewDoc := WordApp.Documents.Add;
  NewDoc.Content := 'Hello World';
  NewDoc.SaveAs('C:\Users\CCR\Documents\Test.doc');
  WordApp.Quit;
end.
```

Technically, a scriptable COM object is one that implements the `IDispatch` interface; assign an `IDispatch` reference to a variant, therefore, and you will be able to invoke the methods and properties it exposes in a so-called 'late bound' fashion.

For a Delphi application that targets OS X, COM does not exist. Thus, while variants are still implemented by the Delphi RTL, the special support for `IDispatch` scripting is not. Since the only `IDispatch` objects available for scripting would be those you wrote yourself, this is not a significant loss — while Microsoft Office for the Macintosh includes a version of OLE Automation internally, it does not expose its `IDispatch` objects to the outside world (you're supposed to use native Mac OS scripting — AppleScript — instead).

Expression engine

The RTL's expression engine allows you to take an arbitrary string and compute a value from it. For example, pass the string `'2 + 2'`, and the number `4` is returned. Evaluated strings can contain numbers, string literals and basic operators (e.g. + and -), together with a small set of intrinsic routines and references to objects of your choice. While this doesn't make for a full scripting engine by any means, it still makes possible dynamically defining and resolving expressions in a nicely generic fashion.

Ultimately, the expression engine is a key part of the LiveBindings system, which connects user interface controls in a FireMonkey application to a database backend. However, the engine can be used independently too, which is way in which we will be looking at it here.

Expression engine basics

Put to work, here's the minimal amount of code necessary to evaluate the string `'1 + 1'` as an expression:

```
uses
  System.Rtti, System.Bindings.EvalProtocol,
  System.Bindings.EvalSys, System.Bindings.Evaluator;

var
  Scope: IScope;
  CompiledExpr: ICompiledBinding;
  Result: TValue;
begin
  Scope := BasicOperators;
  CompiledExpr := Compile('1 + 1', Scope);
  Result := CompiledExpr.Evaluate(Scope, nil, nil).GetValue;
  WriteLn('The result is ', Result.ToString);
end.
```

As shown here, the basic usage pattern is to firstly initialise an `IScope` instance, secondly call the `Compile` function, and thirdly invoke the `Evaluate` method, passing the same `IScope` instance you passed to `Compile`. This returns an instance of the `IValue` interface, on which you call the `GetValue` method to retrieve a `TValue` containing the expression result.

In themselves, the `IScope`, `ICompiledBinding` and `IValue` interfaces are pretty simple:

```
type
  IScope = interface
    ['{DAFE2455-3DB6-40CC-B1D6-1EAC0A29ABEC}']
    function Lookup(const Name: string): IInterface;
  end;

  TCompiledBindingPhaseType = (cbpEmpty,
    cbpPrepareCompilation, cbpCompiling, cbpCompiled,
    cbpPrepareEvaluation, cbpEvaluating, cbpEvaluated,
    cbpEvaluationError);

  ICompiledBinding = interface
    ['{42B9D178-5460-45F8-8CBF-5F8310A4C713}']
    function GetPhase: TCompiledBindingPhaseType;
    function Evaluate(ARoot: IScope;
      ASubscriptionCallback: TSubscriptionNotification;
      Subscriptions: TList<ISubscription>): IValue;
    property Phase: TCompiledBindingPhaseType read GetPhase;
  end;

  IValue = interface
    ['{A495F901-72F5-4384-BA50-EC3B4B42F6C2}']
    function GetType: PTypeInfo;
    function GetValue: TValue;
  end;
```

In the case of `IScope`, this simplicity is rather superficial — the fact the return value of `Lookup` is typed to `IInterface` doesn't really mean implementers can return any interface they like. Consequently, it is best to leave implementing `IScope` to classes defined by the framework itself.

With respect to `ICompiledBinding`, you generally pass to `Evaluate` the same `IScope` instance you passed to `Compile`, and `nil` for the remaining arguments. `Evaluate` can be called multiple times, in which case the expression is evaluated afresh on each call.

If there is a problem with the source expression, an `EEvaluatorError` exception is raised. As a basic debugging aid, you can retrieve a simple breakdown of the expression as it has been parsed by querying for `IDebugBinding` and calling its `Dump` method:

```
var
  CompiledExpr: ICompiledBinding;
begin
  //... code as before
  (CompiledExpr as IDebugBinding).Dump(
    procedure (ALine: string)
    begin
      WriteLn(ALine);
    end);
```

This can be called either before or after `Evaluate` is called. In the case of our `'1 + 1'` example, it outputs the following:

```
Constant table:
  0: 1
  1: 1
  2: $add
Program body:
  0: push 0
  3: push 1
  6: invokedirect 2 2
  11: return
```

The System.Bindings. unit scope*

As indicated by the `uses` clause in the original example, the expression engine is implemented by a number of units in the `System.Bindings.*` unit scope. Designed to be very extensible, the web of units involved can appear quite complex as a result. The important ones are the following:

- `System.Bindings.EvalProtocol` declares the core interface types used by the framework, such as `IScope` and `ICompiledBinding`.
- `System.Bindings.Evaluator` implements the expression compiler and evaluator, exposing a single standalone function called `Compile`. To this you pass the expression to be evaluated, together with an `IScope` implementation that defines the symbols (beyond literals) that the expression may use. On return, the function provides an `ICompiledBinding` instance, against which you call `Evaluate` to actually return the expression result.
- `System.Bindings.EvalSys` defines numerous `IScope` implementations. Two of these are fully implemented, providing support for basic operators (+, -, *, /, =, <>, >=, >, <=, <) and basic constants (`nil`, `True`, `False`, `Pi`). The others are more 'infrastructure'-type things, for example `TNestedScope` and `TDictionaryScope`.
- `System.Bindings.Methods` implements an `IScope` implementation to provide a small set of standard functions for expressions to use.
- `System.Bindings.ObjEval` provides a `WrapObject` function for exposing any given class instance to an expression.
- `System.Bindings.Expression` defines an interface (specifically, an abstract base class) for objects that wrap the lower-level expression engine code.
- `System.Bindings.ExpressionDefaults` provides a standard implementation of this interface.

The last two units here creep more into LiveBindings territory rather than being part of the expression engine as such. While you can still use them outside of a LiveBindings context, there is little point — for the extra layers of code, they abstract little from the underlying interfaces defined in `System.Bindings.EvalProtocol`. For instance, here's the `'1 + 1'` example rewritten to use them:

```
uses
  System.SysUtils, System.Bindings.EvalProtocol,
  System.Bindings.Expression, System.Bindings.ExpressionDefaults;

var
  Expr: TBindingExpression;
  Result: TValue;
begin
  Expr := TBindingExpressionDefault.Create;
  try
    Expr.Source := '1 + 1';
    Expr.Compile([], [], []);
    Result := Expr.Evaluate.GetValue;
    WriteLn('1 + 1 = ', Result.ToString);
  finally
    Expr.Free;
  end;
```

```
end.
```

Expression syntax

The syntax that must be used inside an expression string essentially follows the syntax for actual Delphi code, with a couple of exceptions. The first is that you can delimit string literals using double instead of single quote characters, so long as you are consistent about it — thus, `'this is OK'`, `"so is this"`, `'but this is not"`. The second difference is that any parameterless function exposed to the expression engine *must* appear with a pair of empty brackets when referenced in an expression. For example, say a function called `Foo` is exposed (we will look at the actual mechanics of exposing functions shortly): a valid expression string invoking it must take the form `'Foo()'` rather than simply `'Foo'`. Don't use brackets, and the expression parser will assume the symbol refers to a property or field, leading to a missing identifier exception being raised.

Combining scopes

Without a 'scope', an expression will only be able to contain numeric and string literals. When 'compiled', each expression is therefore associated with one `IScope` instance, which defines the symbols (operators and identifiers) allowed inside the expression.

Since the `Compile` function accepts only one `IScope` instance, what do you do when you want to provide more than one? The answer is to use the `TNestedScope` and `TDictionaryScope` classes.

`TNestedScope` merges two source `IScope` implementations by treating the first as the 'outer' scope and the second as the 'inner' scope. (Imagine a standalone routine in Delphi code: the inner scope will be formed by its local variables and constants, and its outer scope by items defined at the unit level.) If more than two scopes need to be merged, instantiate `TNestedScope` multiple times in a chained fashion:

```
function MergeWithBasicScopes(const MyScope: IScope): IScope;
begin
  Result := TNestedScope.Create(BasicOperators,
            TNestedScope.Create(BasicConstants, MyScope));
end;
```

Here, `MyScope` is the inner-est scope, and therefore takes priority when the same identifier is found in more than one of the source scopes. In the event of an identifier not appearing in `MyScope` but being exposed by both `BasicOperators` and `BasicConstants` (there won't be such a thing in practice, but bear with me), then the one in `BasicConstants` will have priority.

`TDictionaryScope`, in contrast, combines multiple `IScope` implementations into one on the basis of the source scopes being named children of the combined scope. When referenced in an expression, an item from a source scope would then need to be fully qualified (`SourceScopeName.ItemName`).

Enabling standard functions in expressions

The `System.Bindings.Methods` unit implements a series of standard functions for expressions to use: `ToStr`, `ToVariant`, `ToNotifyEvent`, `Round`, `Format`, `UpperCase`, `LowerCase`, `FormatDateTime`, `StrToDateTime`, `Min` and `Max`. These are wrapped into an `IScope` instance created via `TBindingMethodsFactory.GetMethodScope`:

```
uses
  System.Rtti, System.Bindings.EvalProtocol,
  System.Bindings.EvalSys, System.Bindings.Evaluator,
  System.Bindings.Methods;

const
  Expr = '"5 + 5 = " + ToStr(5 + 5) + ", " + UpperCase("ok?")';
var
  Scope: IScope;
  Result: TValue;
begin
  Scope := TNestedScope.Create(BasicOperators,
    TBindingMethodsFactory.GetMethodScope);
  Result := Compile(Expr, Scope).Evaluate(Scope, nil, nil).GetValue;
  WriteLn(Result.ToString); //5 + 5 = 10, OK?
end.
```

For when you only want a subset of the standard functions available, `GetMethodScope` is overloaded to accept an open array of function names:

```
//only make available Min, Max and Round
Scope := TBindingMethodsFactory.GetMethodScope(['Min', 'Max', 'Round']);
```

Another way to limit the standard set is to call the `UnregisterMethod` class procedure, passing the name of the function to remove, and call the parameterless version of `GetMethodScope` as before.

Adding to the standard functions

Alongside `UnregisterMethod`, `TBindingMethodsFactory` also has a `RegisterMethod` class procedure. To this you pass an instance of the `TMethodDescription` record, which itself is constructed from a number of items. Here's how you might go about exposing the `CompareText` function from `System.SysUtils`:

```
uses
  System.SysUtils, System.Bindings.EvalProtocol,
  System.Bindings.Methods, System.Bindings.Consts;

function CompareTextWrapper: IInvokable;
begin
  Result := MakeInvokable(
    function(Args: TArray<IValue>): IValue
    begin
      //confirm two arguments were passed
      if Length(Args) <> 2 then
        raise EEvaluatorError.CreateResFmt(
          @SUnexpectedArgCount, [2, Length(Args)]);
      //do the actual comparision and return the result
      Result := TValueWrapper.Create(CompareText(
        Args[0].GetValue.ToString,
        Args[1].GetValue.ToString, loUserLocale));
    end)
end;

begin
  TBindingMethodsFactory.RegisterMethod(
    TMethodDescription.Create(CompareTextWrapper,
      'CompareText', '', '', True, '', nil));
```

The `TMethodDescription` constructor has the following signature:

```
constructor Create(const AInvokable: IInvokable;
  const AID, AName, AUnitName: string; ADefaultEnabled: Boolean;
  const ADescription: string; AFrameworkClass: TPersistentClass);
```

`IInvokable` is a simple interface defined by `System.Bindings.EvalProtocol` as thus:

```
  IInvokable = interface
    ['{0BB8361C-AAC7-42DB-B970-5275797DF41F}']
    function Invoke(const Args: TArray<IValue>): IValue;
  end;
```

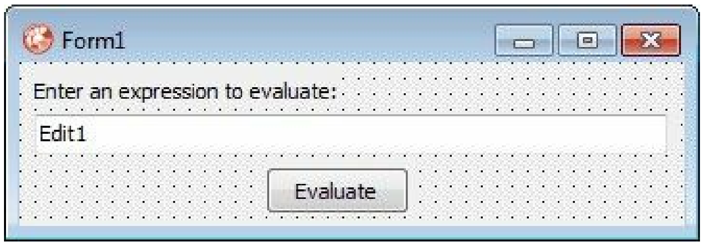
The same unit provides the `TValueWrapper` class for wrapping a `TValue` into an `IValue` interface (the added abstraction is to allow 'locations' rather than just 'values' to be returned, though this is irrelevant in our case). `System.Bindings.Methods` itself then provides the `MakeInvokable` routine to create an `IInvokable` implementation from an anonymous method, as used in the previous example.

The third, fourth and sixth parameters to `TMethodDescription.Create` are only used with the full `LiveBindings` system. Passing `True` for the fifth parameter means the routine will be incorporated into the `IScope` object returned by the parameterless version of `GetMethodScope`, otherwise it only gets added when the array version of `GetMethodScope` is used, and the caller explicitly names the custom function.

Exposing objects

If truth be told, exposing an existing function using `TBindingMethodsFactory` is a little tedious. Thankfully, alongside it stands the `WrapObject` function of `System.Bindings.ObjEval`: pass this routine an object of your choice, and an `IScope` implementation is returned that exposes almost everything that can be retrieved from the object using RTTI. Assuming the default RTTI settings are in operation, that means practically every public instance method and property, together with every field (class methods are not exposed however). Moreover, in the case of an object with child objects, those are automatically exposed too.

To see this in effect, create either a new VCL or FireMonkey HD application, then place a `TLabel`, `TEdit` and `TButton` on the form:



Next, handle the button's `OnClick` event like this:

```
uses
  System.Bindings.EvalProtocol, System.Bindings.Evaluator,
  System.Bindings.EvalSys, System.Bindings.ObjEval,
  System.Bindings.Methods;

procedure TForm.Button1Click(Sender: TObject);
var
  Objects: TDictionaryScope;
  Scope: IScope;
  Output: IValue;
begin
  Objects := TDictionaryScope.Create;
  Objects.Map.Add('Application', WrapObject(Application));
  Objects.Map.Add('Screen', WrapObject(Screen));
  Scope := TNestedScope.Create(BasicOperators,
             TNestedScope.Create(BasicConstants,
               TNestedScope.Create(
               TBindingMethodsFactory.GetMethodScope,
                 TNestedScope.Create(Objects, WrapObject(Self)))));
  Output := Compile(Edit1.Text, Scope).Evaluate(Scope, nil, nil);
  ShowMessage(Output.GetValue.ToString);
end.
```

Here, we expose the basic operators and constants provided by `System.Bindings.EvalSys` as the outer-est scope and second most outer scope respectively, followed by the standard functions provided by `TBindingMethodsFactory`, the `Application` and `Screen` objects defined by the relevant `Forms` unit (either `FMX.Forms` or `VCL.Forms`), and finally, the members of the form itself, which will be callable by an expression without fully qualifying with the form's name and a full stop.

On running the application, you should be able to enter any of the following expressions into the edit box, and it work as expected:

```
"The main form's caption is " + Application.MainForm.Caption

"You've just clicked a button labelled '" + Button1.Text + "'"

"The same button is called '" + Application.MainForm.Button1.Name + "'"

'The first (and only?) form of the application has ' + ToStr(Screen.Forms[0].ComponentCount) + ' controls'

"The name of this form's first control is '" + Components[0].Name + "', and its text is '" + Components[0].Text + "'"
```

Notice how child objects are exposed in a 'dynamic', script-like fashion — the third example here wouldn't be legal in compiled code, given `Button1` is an object in your own form, not the base class that `Application.MainForm` is typed to.

Expression engine quirks: brackets vs. no brackets

Consider the following code, which will raise an `EEvaluatorError` exception:

```
uses
  System.SysUtils, System.Bindings.EvalProtocol,
  System.Bindings.EvalSys, System.Bindings.Methods,
  System.Bindings.Evaluator, System.Bindings.ObjEval;

const
  Expr = 'DateTimeToStr(Now)';
var
  Scope: IScope;
  Output: IValue;
begin
  Scope := TBindingMethodsFactory.GetMethodScope;
  Output := Compile(Expr, Scope).Evaluate(Scope, nil, nil);
  WriteLn(Output.GetValue.ToString);
end.
```

The fact an exception is raised shouldn't be surprising, since neither `DateTimeToStr` nor `Now` are amongst the set of functions implemented by `TBindingMethodsFactory`.

So, let’s implement them ourselves with a helper class and `WrapObject`:

```
type
  TDateTimeUtils = class
    function DateTimeToStr(const ADateTime: TDateTime): string;
    function Now: TDateTime;
  end;

function TDateTimeUtils.DateTimeToStr(
  const ADateTime: TDateTime): string;
begin
  Result := System.SysUtils.DateTimeToStr(ADateTime);
end;

function TDateTimeUtils.Now: TDateTime;
begin
  Result := System.SysUtils.Now;
end;

const
  Expr = 'DateTimeToStr(Now)';
var
  ScopeSource: TDateTimeUtils;
  Scope: IScope;
  Output: IValue;
begin
  ScopeSource := TDateTimeUtils.Create;
  try
    Scope := TNestedScope.Create(
      TBindingMethodsFactory.GetMethodScope,
      WrapObject(ScopeSource));
    Output := Compile(Expr, Scope).Evaluate(Scope, nil, nil);
    WriteLn(Output.GetValue.ToString);
  finally
    ScopeSource.Free;
  end;
end.
```

Run the revised application, and it should now go through without exceptions. However, 30/12/1899 (or 12/30/1899 if you speak American) will be returned!

This is caused by a combination of two things: the expression parser requires parameterless functions to be called with a pair of empty brackets, and the evaluator’s ‘lazy lookup’ of object members returns `nil` instead of raising an exception when the source expression references a member that doesn’t exist.

In our case, the problem can be fixed in one of two ways: ensure the source expression uses brackets (i.e., require `'DateTimeToStr(Now())'` not `'DateTimeToStr(Now)'`), or make `Now` a property rather than a parameterless function:

```
type
  TDateTimeUtils = class
  strict private
    function GetNow: TDateTime;
  public
    function DateTimeToStr(const ADateTime: TDateTime): string;
    property Now: TDateTime read GetNow;
  end;

function TDateTimeUtils.GetNow: TDateTime;
begin
  Result := System.SysUtils.Now;
end;
```

Once made a property, brackets now cannot be used when referring to it inside an expression. While this is consistent with Delphi itself (you can’t use `MyForm.Caption()` to retrieve the form’s caption, for example), it has the implication of there being no way to be liberal and allow both with or without brackets.

12. Runtime type information (RTTI)

‘Runtime type information’ (RTTI) is the mechanism by which information about the types used by your program can be retrieved at runtime. It is also the system by which you can call methods and change property and field values without knowing at compile-time what methods and properties exactly you will be calling or changing, e.g. via name strings filled out at runtime.

For historical reasons, Delphi has two main ‘levels’ of RTTI support: basic RTTI, which was there from the beginning, and ‘extended’ RTTI, which was introduced only more recently. ‘Basic’ RTTI covers as much as is necessary for the core VCL, which is to say, quite a bit for its time, but a pale shadow of the ‘reflection’ or ‘introspection’ capabilities of popular managed languages. ‘Extended’ RTTI substantially corrects this, though at the cost of producing executables that are comparatively ‘bulked up’ with all the extra type information. For this reason, the amount of extended RTTI produced is configurable; in the following pages, I will assume the (reasonably comprehensive) defaults.

The basic RTTI interface

If your reflection needs are small or very specific, the original RTTI interface can be enough. Even if they aren't, having a basic grasp of basic RTTI can be helpful for understanding its higher-level sibling, which essentially wraps and extends it, rather than being a totally separate system.

Basic RTTI is based around pointers to `TTypeInfo` records, one for each type, that you retrieve via the `TypeInfo` standard function. For example, use `TypeInfo(AnsiString)` to return the type information for `AnsiString`, and in the context of a generic method body, `TypeInfo(T)` to return the information for parameterised type called `T`. In itself, `TypeInfo` returns an untyped pointer, which must be assigned to a variable typed to `PTypeInfo` to become useful.

`PTypeInfo` and related types are exported by the `System.TypInfo` unit (note the missing 'e'), and has the following declaration:

```
type
  TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration,
    tkFloat, tkString, tkSet, tkClass, tkMethod, tkWChar,
    tkLString, tkWString, tkVariant, tkArray, tkRecord,
    tkInterface, tkInt64, tkDynArray, tkUString, tkClassRef,
    tkPointer, tkProcedure);

  PTypeInfo = ^TTypeInfo;
  TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
  end;
```

The element names of the enumeration should be generally self-explanatory, though watch out for the fact `tkString` means a `ShortString`, `tkLString` an `AnsiString` and `tkUString` a `UnicodeString`/`string`.

Even just this amount of RTTI can be useful with generics, since it allows implementing runtime constraints on parameterised types where the compile-time constraints system is not fine grained or flexible enough:

```
uses
  System.SysUtils, System.TypInfo;

type
  TMyGeneric<T> = class
    constructor Create;
  end;

resourcestring
  SConstraintError = 'TMyGeneric can only be instantiated ' +
    'with a class or interface type';

constructor TMyGeneric<T>.Create;
var
  Info: PTypeInfo;
begin
  inherited;
  Info := TypeInfo(T);
  if not (Info.Kind in [tkClass, tkInterface]) then
    raise EInvalidCast.Create(SConstraintError);
end;

var
  OK: TMyGeneric<TStringBuilder>;
  NotOK: TMyGeneric<Integer>;
begin
  OK := TMyGeneric<TStringBuilder>.Create; //acceptable
  NotOK := TMyGeneric<Integer>.Create;     //causes exception
end.
```

Using GetTypeData

Once you have a pointer to a `TTypeInfo` record, the next step is typically to retrieve a pointer to a `TTypeData` structure. This is done via the `GetTypeData` function, also declared in `System.TypInfo`:

```
var
  Info: PTypeInfo;
  Data: PTypeData;
begin
  Info := TypeInfo(string);
  WriteLn(Info.Name);
  Data := GetTypeData(Info);
  //use Data...
```

`TTypeData` contains numerous fields which have validity depending upon the kind of type that is being described. Be careful not to access a field that isn't relevant to the type (e.g., `MaxValue` in the case of a floating point number) — due to the low-level way `TTypeData` is implemented, the compiler won't stop you doing such a thing, but a rubbish request will get rubbish out:

- For an `AnsiString`, the `CodePage` field returns the compile-time codepage the type was explicitly or implicitly declared with. For instance, `GetTypeData(TypeInfo(UTF8String)).CodePage` will return 65001, i.e. the value of the `CP_UTF8` constant.
- For a floating point number, the `FloatType` field reports the underlying fundamental type, expressed as a value from the `TFloatType` enumeration (`ftSingle`, `ftDouble`, `ftExtended`, `ftComp`, or `ftCurr`).
- For a class, `PropCount` returns the total number of published properties (including published events) declared in the class and its ancestors, and `UnitName` the name of the unit the class is declared in.
- For an object interface, `GUID` returns, you've guessed it, the interface's GUID:

```
uses
  System.SysUtils, System.TypInfo;

type
  ISomeIntf = interface
  ['{9EB345D5-86DA-4D59-82A3-436F24382840}']
  end;

var
  GUID: TGUID;
  S: string;
begin
  GUID := GetTypeData(TypeInfo(ISomeIntf)).Guid;
  S := GUID.ToString; //{9EB345D5-86DA-4D59-82A3-436F24382840}
```

- For an ordinal type, `OrdType` returns the underlying type expressed as a value from the `TOrdType` enumeration (`otSByte`, `otUByte`, `otSWord`, `otUWord`, `otSLong` or `otULong`), `MinValue` returns the lowest ordinal value an instance can legitimately have, and `MaxValue` the highest.

Here's an example of `GetTypeData` being used to discover information about a group of ordinal types:

```
uses
  System.TypInfo;

procedure WriteOrdinalTypeInfo(const Info: PTypeInfo);
var
  Data: PTypeData;
begin
  WriteLn(Info.Name);
  if not (Info.Kind in [tkInteger, tkChar, tkEnumeration, tkWChar]) then
  begin
    WriteLn('Not an ordinal type!');
    Exit;
  end;
  Data := GetTypeData(Info);
  WriteLn('Min ordinal value = ', Data.MinValue);
  WriteLn('Max ordinal value = ', Data.MaxValue);
  Write('Stored as ');
  case Data.OrdType of
    otSByte: WriteLn('a signed byte value');
    otUByte: WriteLn('an unsigned byte value');
    otSWord: WriteLn('a signed 2-byte value');
    otUWord: WriteLn('an unsigned 2-byte value');
    otSLong: WriteLn('a signed 4-byte value');
    otULong: WriteLn('an unsigned 4-byte value');
  end;
  WriteLn;
end;

type
  TMyEnum = (First, Second, Third);

begin
  WriteOrdinalTypeInfo(TypeInfo(Boolean));
  WriteOrdinalTypeInfo(TypeInfo(Char));
  WriteOrdinalTypeInfo(TypeInfo(Integer));
  WriteOrdinalTypeInfo(TypeInfo(TMyEnum));
```

```
end.
```

This program will output the following:

```
Boolean
Min ordinal value = 0
Max ordinal value = 1
Stored as an unsigned byte value

Char
Min ordinal value = 0
Max ordinal value = 65535
Stored as an unsigned 2-byte value

Integer
Min ordinal value = -2147483648
Max ordinal value = 2147483647
Stored as a signed 4-byte value

TMyEnum
Min ordinal value = 0
Max ordinal value = 2
Stored as an unsigned byte value
```

The `System.TypInfo` unit also provides a couple of functions for converting enumerated values to and from a string, `GetEnumName` and `GetEnumValue` — please see chapter 2 for examples of how to use them — together with a large group of routines for getting and setting published properties. The latter work via yet another pointer structure, `PPropInfo`, though in this case more as a 'magic handle' than something you inspect directly:

```
uses
  System.SysUtils, System.Classes, System.TypInfo;

type
  TSomeComponent = class(TComponent)
  strict private
    FText: string;
  published
    property Text: string read FText write FText;
  end;

var
  Obj: TSomeComponent;
  PropInfo: PPropInfo;
  S: string;
begin
  Obj := TSomeComponent.Create(nil);
  //set the Text property dynamically
  PropInfo := GetPropInfo(Obj, 'Text');
  SetStrProp(Obj, PropInfo,
    'The low level RTTI interface requires concentration!');
  //get the Text property dynamically
  S := GetStrProp(Obj, PropInfo);
  WriteLn(S);
  Obj.Free;
end.
```

Once you're in the realm of properties, you would usually do better to use the higher-level interface of the `System.Rtti` unit instead, so that's what we'll turn to now.

The extended RTTI interface

The `System.Rtti` unit is centred on the `TRttiContext` record type, which has the following public members:

```
//pseudo-constructor and pseudo-destructor
class function Create: TRttiContext; static;
procedure Free;
//other
function GetType(ATypeInfo: Pointer): TRttiType; overload;
function GetType(AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType(const AQualifiedName: string): TRttiType;
function GetPackages: TArray<TRttiPackage>;
```

Put to work, you usually just declare a variable typed to `TRttiContext` and call its methods straight away:

```
var
  Context: TRttiContext;
  LType: TRttiType;
begin
  LType := Context.FindType('System.Integer');
  //work with LType...
  for LType in Context.GetTypes do
    //work with LType once more...
```

While a `Create` pseudo-constructor and `Free` pseudo-destructor are defined, calling either is strictly optional. Under the bonnet, all `TRttiContext` records ultimately delegate to a shared 'master' object that gets created when necessary, and freed when the last `TRttiContext` instance goes out of scope. Using `Create`, therefore, merely ensures this master object is initialised immediately (if it isn't already), and calling `Free` just decrements its internal reference count ahead of the current `TRttiContext` record going out of scope.

The `TRttiType` objects returned by `GetType`, `GetTypes` and `FindType` are, roughly speaking, higher level versions of `PTypeInfo` and `PTypeData`. You should never try and instantiate `TRttiType` or a descendant class directly — rather, just poll a `TRttiContext` record for an instance. Moreover, no `TRttiType` object (or for that matter, any other object returned directly or indirectly from a `TRttiContext` record) should be explicitly freed either. This is because it will be 'owned' by the internal master object previously mentioned, to be freed automatically when that master object is freed.

Using TRttiContext.FindType

A limitation of `FindType` is that it only works with so-called 'public' types, meaning types declared in the `interface` section of a unit. Types declared in the `implementation` section, or in the DPR, will not be findable, though it will still be possible to get type information objects for them using `GetType`.

If and when you call `FindType`, ensure you pass it a fully qualified identifier — in other words, include the name of the unit (including the unit scope, if defined) and a full stop. For example, passing `'System.Classes.TStringList'` will work, but just `'TStringList'` (or indeed `'Classes.TStringList'`) will not; in the case of a type built into the language, assume `System` is the parent unit (e.g., `'System.string'`).

If a type isn't found, `nil` will be returned. If you're finding this happening when you're sure the type does exist, then the type probably hasn't been explicitly referenced anywhere outside of the `FindType` call. In such a case, the compiler won't be able to tell the type is being used, and so ignore it when linking different parts of the final executable together (from the compiler's point of view, a string is a string, not a type reference). The workaround is to explicitly reference the type somewhere, e.g. in a dummy `initialization` section of its unit:

```
unit MyDynamicallyRefedClass;

interface

type
  TMyDynamicallyRefedObject = class
    //stuff...
  end;

implementation

initialization
  TMyDynamicallyRefedObject.ClassName;
end.
```

RTTI and packages

RTTI in general, and `TRttiContext` specifically, work across package boundaries, whether the packages in question are ones the application was built with or have been loaded dynamically at runtime.

As a result, using packages will affect the number of objects `TRttiContext.GetTypes` returns. In the case of runtime package linked to at compile-time, a packaged type will be findable so long as its unit is used by the program somewhere — in other words, types will be brought in at the *unit* level rather than the individual *type* level. In the case of a package linked to at runtime, in contrast, packaged types will be findable as soon as the package itself is loaded (i.e., types are brought in at the *package* level).

Be warned `TRttiContext.GetPackages` is slightly misnamed. This is because it gives a breakdown of types not by package, but by module. Consequently, it always treats the main executable as the first 'package'. In the case of an unpackaged application, therefore, `GetPackages` will return not `nil` but an array one element long.

`TRttiPackage` itself has the following public members:

```
property BaseAddress: Pointer read FBaseAddress;
property Handle: HMODULE read GetHandle;
property Name: string read GetName;
function GetTypes: TArray<TRttiType>;
function FindType(const AQualifiedName: string): TRttiType;
```

`Name` returns the file name of the package (including its path), `Handle` something you can pass to functions like `GetPackageDescription` and `GetProcAddress`, and `GetTypes` and `FindType` work as they do on `TRttiContext`, only in the context of the specific module's types.

TRttiType and its descendants

Rather than overlaying fields relevant to different sorts of type à la `TTypeData`, `TRttiType` is a class that exposes information for different sorts of thing via descendants you can query for. Thus, if you retrieve the type information object for a certain record type, an instance of `TRttiRecordType` will be returned, and for a set type, `TRttiSetType` and so on —

```
var
  Context: TRttiContext;
  LType: TRttiType;
begin
  for LType in Context.GetTypes do
    if LType is TRttiRecordType then
      //use TRttiRecordType(LType)
    else if LType is TRttiSetType then
      //use TRttiSetType(LType)
```

The class hierarchy involved is as follows:

```
TRttiType
  —— TRttiOrdinalType
       —— TRttiEnumerationType
  —— TRttiInt64Type
  —— TRttiFloatType
  —— TRttiPointerType
  —— TRttiSetType
  —— TRttiStringType
       —— TRttiAnsiStringType
  —— TRttiArrayType
  —— TRttiDynamicArrayType
  —— TRttiInvokableArrayType
  —— TRttiClassRefType
  —— TRttiStructuredType
       —— TRttiInstanceType
       —— TRttiInterfaceType
       —— TRttiRecordType
```

The naming is generally straightforward, though classes are referred to as 'instance' types and metaclasses as 'class ref' types.

TRttiType members

The core members of `TRttiType` are as thus:

```
function GetAttributes: TArray<TCustomAttribute>;
property BaseType: TRttiType read GetBaseType;
property Handle: PTypeInfo read GetHandle;
property IsManaged: Boolean read GetIsManaged;
property IsPublicType: Boolean read GetIsPublicType;
property QualifiedName: string read GetQualifiedName;
property Name: string read GetName; //i.e., the unqualified name
property Package: TRttiPackage read FPackage;
property TypeKind: TTypeKind read GetTypeKind;
property TypeSize: Integer read GetTypeSize;
```

`BaseType` returns the type information object for the immediate ancestor type, if one exists (the parent class for a class, the parent interface type for an interface), otherwise `nil`. Of the other properties, `Handle` returns a `PTypeInfo` pointer to the low-level RTTI record for the described type, `IsManaged` returns `True` if it is a managed type like `string` or `Variant`, `IsPublic` returns `True` if it is declared in the `interface` section of a unit, `QualifiedName` includes the unit name for public types, and `TypeSize` returns the size in bytes of an instance of the type. In the case of a reference type (`string`, `TBytes`, `IInterface`, etc.), that will be the size of a pointer (4 bytes when targeting OS X or Win32, 8 when targeting Win64).

Beyond these core members, `TRttiType` also provides some helper properties for the most common sorts of described type. This is to make testing and casting a bit easier:

```
property IsInstance: Boolean read GetIsInstance;
property AsInstance: TRttiInstanceType read GetAsInstance;
property IsOrdinal: Boolean read GetIsOrdinal;
property AsOrdinal: TRttiOrdinalType read GetAsOrdinal;
property IsRecord: Boolean read GetIsRecord;
property AsRecord: TRttiRecordType read GetAsRecord;
property IsSet: Boolean read GetIsSet;
property AsSet: TRttiSetType read GetAsSet;
```

In each case, `LType.IsXXX` is just a shortcut for `LType is TRttiXXXType`, and `LType.AsXXX` just a shortcut for `(LType as TRttiXXXType)`.

The final group of methods exposed by `TRttiType` are essentially refugees from `TRttiStructuredType`, moved upstream to `TRttiType` for convenience:

```
{ all }
function GetMethods: TArray<TRttiMethod>; overload;
function GetFields: TArray<TRttiField>;
function GetProperties: TArray<TRttiProperty>;
function GetIndexedProperties: TArray<TRttiIndexedProperty>;
{ specific member (GetMethods as well as GetMethod in case of
  overloading) }
function GetMethod(const AName: string): TRttiMethod;
function GetMethods(const AName: string): TArray<TRttiMethod>; overload;
function GetField(const AName: string): TRttiField;
function GetProperty(const AName: string): TRttiProperty;
function GetIndexedProperty(const AName: string): TRttiIndexedProperty;
{ introduced in the current type only }
function GetDeclaredMethods: TArray<TRttiMethod>;
function GetDeclaredProperties: TArray<TRttiProperty>;
function GetDeclaredFields: TArray<TRttiField>;
function GetDeclaredIndexedProperties: TArray<TRttiIndexedProperty>;
```

The difference between the `GetDeclaredXXX` and `GetXXX` methods is that the former only return the members introduced by the type itself, whereas `GetXXX` return members introduced by ancestor types as well:

```
type
  TMyObject = class(TObject)
    procedure Foo; virtual; abstract;
  end;

var
  Context: TRttiContext;
  AType: TRttiType;
begin
  AType := Context.GetType(TMyObject);
  WriteLn(Length(AType.GetMethods));         //output: 34
  WriteLn(Length(AType.GetDeclaredMethods)); //output: 1
```

In this case, `GetMethods` returns 34 items since `TObject` (the parent class of `TMyObject`) introduces 33 methods itself.

Items are returned in order, so that newly-introduced members come first, those introduced by the parent type next, those by the grandparent type after that, and so on. This allows looking up a method, field or property in the manner the compiler would do, which is to always go for the item declared deepest into the inheritance hierarchy if and when a name clash arises.

Here's a simple example of these methods in use:

```
uses
```

```
  System.Rtti;

type
  TTest = class
    MyField: Integer;
    procedure Foo(const Arg: string);
  end;

procedure TTest.Foo(const Arg: string);
begin
  WriteLn('Arg = ', Arg);
  WriteLn('MyField = ', MyField);
end;

var
  Context: TRttiContext;
  LType: TRttiType;
  LMethod: TRttiMethod;
  Instance: TTest;
begin
  LType := Context.GetType(TTest);
  WriteLn('No. of fields in TTest: ', Length(LType.GetFields));
  WriteLn('No. of public methods declared by TTest: ',
    Length(LType.GetDeclaredMethods));
  WriteLn('Total no. of public methods on TTest: ',
    Length(LType.GetMethods));
  Instance := TTest.Create;
  try
    LType.GetField('MyField').SetValue(Instance, 42);
    LMethod := LType.GetMethod('Foo');
    LMethod.Invoke(Instance, ['Hello RTTI world']);
  finally
    Instance.Free;
  end;
end.
```

This example queries a class type. When used with a record type the syntax is very similar, however it is not identical. The first difference is that the initial appeal to `GetType` must be slightly extended with an explicit call to `TypeInfo`. This is because record types are not tangible entities like metaclasses:

```
LType := Context.GetType(TypeInfo(TTest));
```

Secondly, `TRttiField.SetValue` requires the subject record to be explicitly referenced with the @ sign:

```
LType.GetField('MyField').SetValue(@Instance, 42);
```

The same thing would apply to `TRttiProperty.SetValue`. Lastly, `TRttiMethod.Invoke` requires a record to be explicitly boxed into a `TValue` instance:

```
LMethod.Invoke(TValue.From(Instance), ['Hello RTTI world']);
```

Slightly special handling for `Invoke` is also required in the case of a class method or constructor for a class. This is because the first argument passed to `Invoke` represents the implicit `Self` parameter, which in the case of a class method should be a reference to the class not an instance of the class:

```
uses
  System.Classes, System.Rtti;

type
  TClassyObject = class
    class function Foo: string;
  end;

class function Foo: string;
begin
  Result := 'My name is ' + ClassName;
end;

var
  Context: TRttiContext;
  LType: TRttiInstanceType;
  LMethod: TRttiMethod;
begin
  LType := Context.FindType('TClassyObject').AsInstance;
  LMethod := LType.GetMethod('Foo');
  WriteLn(LMethod.Invoke(LType.MetaclassType, []).ToString);
```

In the case of a *static* class method you can pass what you like though, since there is no implicit `Self` parameter in that situation to set up.

More on TRttiField, TRttiProperty, TRttiIndexedProperty and TRttiMethod

All of `TRttiField`, `TRttiProperty`, `TRttiIndexedProperty` and `TRttiMethod` inherit from `TRttiMember`. This class introduces `Name`, `Parent` and `Visibility` properties: `Name` is self-explanatory, `Parent` gets you a reference to the `TRttiType` object from which the member came from, and `Visibility` is an enumeration with the possible values `mvPrivate`, `mvProtected`, `mvPublic` and `mvPublished` (the enumerated type itself, `TMemberVisibility`, is declared in `System.TypInfo`).

`TRttiField` adds the following:

```
property FieldType: TRttiType read GetFieldType;
property Offset: Integer read GetOffset;
function GetValue(Instance: Pointer): TValue;
procedure SetValue(Instance: Pointer; const AValue: TValue);
```

`FieldType` gives you a type information object for what the field is typed to — in the case of a field typed to `string`, it will therefore return the `TRttiType` object for the `string` type — and `Offset` the number of bytes the field in memory is placed following the start of an instance of the type.

`TRttiProperty` has a similar look to `TRttiField`:

```
property PropertyType: TRttiType read GetPropertyType;
function GetValue(Instance: Pointer): TValue;
procedure SetValue(Instance: Pointer; const AValue: TValue);
property IsReadable: Boolean read GetIsReadable;
property IsWritable: Boolean read GetIsWritable;
```

`TRttiIndexedProperty` is similar again, though with a few more members:

```
property PropertyType: TRttiType read GetPropertyType;
property ReadMethod: TRttiMethod read GetReadMethod;   //getter
property WriteMethod: TRttiMethod read GetWriteMethod; //setter
function GetValue(Instance: Pointer;
  const Args: array of TValue): TValue;
procedure SetValue(Instance: Pointer;
  const Args: array of TValue; const Value: TValue);
property IsReadable: Boolean read GetIsReadable;
property IsWritable: Boolean read GetIsWritable;
property IsDefault: Boolean read GetIsDefault;
```

Given an array property cannot be directly mapped to a field, `ReadMethod` will always be valid if `IsReadable` returns `True` and `WriteMethod` will always be valid if `IsWriteable` returns `True`. Further, both `GetValue` and `SetValue` take an array of index specifiers since an array property can have an arbitrary number of 'dimensions'. In the most common case of a single-dimensional property that has an integral indexer, just provide a single number:

```
uses
  System.Classes, System.Rtti;

var
  Context: TRttiContext;
  LType: TRttiType;
  Obj: TStringList;
begin
  Obj := TStringList.Create;
  try
    Obj.CommaText := 'Zero,One,Two,Three';
    LType := Context.GetType(Obj.ClassType);
    WriteLn(LType.GetIndexedProperty('Strings').GetValue(
      Obj, [1]).ToString); //output: One
  finally
    Obj.Free;
  end;
end.
```

`TRttiMethod` once again follows a similar (or at least similar-ish) pattern:

```
TRttiParameter = class(TRttiNamedObject)
  //skipping private members...
  property Flags: TParamFlags read GetFlags;
  property ParamType: TRttiType read GetParamType;
end;

TDispatchKind = (dkStatic, dkVtable, dkDynamic, dkMessage, dkInterface);
```

```
TRttiMethod = class(TRttiMember)
  //skipping private members...
  function Invoke(Instance: TObject;
    const Args: array of TValue): TValue;
  function Invoke(Instance: TClass;
    const Args: array of TValue): TValue;
  function Invoke(Instance: TValue;
    const Args: array of TValue): TValue;

  function GetParameters: TArray<TRttiParameter>;
  property ReturnType: TRttiType read GetReturnType;
  property HasExtendedInfo: Boolean read GetHasExtendedInfo;

  property IsConstructor: Boolean read GetIsConstructor;
  property IsDestructor: Boolean read GetIsDestructor;
  property IsClassMethod: Boolean read GetIsClassMethod;
  property IsStatic: Boolean read GetIsStatic;
  property MethodKind: TMethodKind read GetMethodKind;

  //low level stuff...
  function CreateImplementation(AUserData: Pointer;
    const ACallback: TMethodImplementationCallback):
    TMethodImplementation;
  property CallingConvention: TCallConv read GetCallingConvention;
  property CodeAddress: Pointer read GetCodeAddress;
  property DispatchKind: TDispatchKind read GetDispatchKind;
  property VirtualIndex: Smallint read GetVirtualIndex;
end;
```

`TMethodKind` and `TCallConv` are defined in `System.TypInfo` as thus:

```
TMethodKind = (mkProcedure, mkFunction, mkConstructor,
  mkDestructor, mkClassProcedure, mkClassFunction,
  mkClassConstructor, mkClassDestructor, mkOperatorOverload,
  mkSafeProcedure, mkSafeFunction);

TCallConv = (ccReg, ccCdecl, ccPascal, ccStdCall, ccSafeCall);
```

The ‘calling convention’ defines the way parameters and (if applicable) result value are passed and returned. The five elements of `TCallConv` correspond to the five calling conventions supported by the 32 bit Delphi compiler, the ‘register’ convention (alias ‘Borland fastcall’) being the default. Since `Invoke` handles the differences between calling conventions for you, the `CallingConvention` property is really just for info — a point that goes for the `CodeAddress`, `DispatchKind` and `VirtualIndex` properties too.

TRttiXXXType classes

Of the descendant classes of `TRttiType`, `TRttiStructuredType` adds nothing due to the `GetXXX` and `GetDeclaredXXX` methods being on `TRttiType` itself. Otherwise, public members are introduced as follows:

- `TRttiRecordType` adds a `ManagedFields` property, which returns an array of `TRttiManagedField` objects that themselves expose `FieldType` and `FieldOffset` properties. A ‘managed’ field means a field that has a managed type, such as `string`, `Variant`, etc. As such, you can also discover whether the record type has any of these by inspecting the inherited `GetFields` method, though doing a quick check for `nil` against `ManagedFields` will be quicker if all you want to know is whether the type has at least one managed field, for example to discover whether it is safe to call `FillChar` or `Move` on instances of it.

- `TRttiInstanceType` adds `DeclaringUnitName`, `MetaclassType` and `VMTSize` properties, together with `GetDeclaredImplementedInterfaces` and `GetImplementedInterfaces` methods that return an array of `TRttiInterfaceType` objects. Of the properties, `MetaclassType` the metaclass of the type being described — don’t confuse it with `ClassType`, which returns the metaclass for `TRttiInstanceType` itself:

```
  uses System.SysUtils, System.Rtti;

  var
    Context: TRttiContext;
    InstType: TRttiInstanceType;
    S1, S2: string;
  begin
    InstType := Context.GetType(TStringBuilder).AsInstance;
    S1 := InstType.ClassType.ClassName;     //TRttiInstanceType
    S2 := InstType.MetaclassType.ClassName; //TStringBuilder
```

`VMTSize` returns the size in bytes of the described class’ ‘virtual method table’. Despite its name, the VMT of a Delphi class actually holds pointers to a few more things beyond its virtual methods (e.g., it contains a pointer to the class' own name). This causes `VMTSize` in XE2 to equal 11 + the total number of virtual methods across the class and its descendants, multiplied by the size of a pointer.

- `TRttiInterfaceType` adds `GUID`, `IntfFlags` and `DeclaringUnitName` properties. If the interface hasn't been given a GUID, the `GUID` property will return the value of `TGUID.Empty` (i.e., `00000000-0000-0000-0000-000000000000`) `IntfFlags` is typed to `TIntfFlags`, a set declared in `System.TypInfo` unit with the possible values `ifHasGuid`, `ifDispInterface` and/or `ifDispatch`.

- `TRttiClassRefType` adds `InstanceType` and `MetaclassType` properties, the former returning the corresponding `TRttiInstanceType` object and the latter the `TClass` reference of the metaclass described.

- `TRttiOrdinalType` adds `OrdType`, `MinValue` and `MaxValue` properties. `OrdType` is typed to the `TOrdType` enumeration we met when looking at the `System.TypInfo` unit; the other two are typed to `LongInt`. Thus, `MaxValue` for the `Boolean` type, for example, will return 1:

```
var
  Context: TRttiContext;
  OrdType: TRttiOrdinalType;
begin
  OrdType := Context.GetType(TypeInfo(Boolean)).AsOrdinal;
  WriteLn(OrdType.MinValue); //output: 0
  WriteLn(OrdType.MaxValue); //output: 1
```

- `TRttiEnumerationType` adds to `TRttiOrdinalType` an `UnderlyingType` property. This returns the type information object for the base type in the case of a sub-range type, otherwise `Self` is returned, in effect:

```
type
  TMyEnum = (One, Two, Three);
  TMySubRange = One..Two;

var
  Context: TRttiContext;
  LType: TRttiEnumerationType;
begin
  LType := Context.GetType(
    TypeInfo(TMyEnum)) as TRttiEnumerationType;
  WriteLn(LType = LType.UnderlyingType); //output: TRUE

  LType := Context.GetType(
    TypeInfo(TMySubRange)) as TRttiEnumerationType;
  WriteLn(LType = LType.UnderlyingType); //output: FALSE
  WriteLn(LType.Name, ' is a sub-range type of ',
    LType.UnderlyingType.Name);          //output: TMyEnum
end.
```

- `TRttiInt64Type` adds `MinValue` and `MaxValue` properties which aren't in themselves very interesting — just use `Low(Int64)` and `High(Int64)` respectively.

- `TRttiFloatType` adds a `FloatType` property typed to the `TFloatType` enumeration of `System.TypInfo`:

```
Info := Context.GetType(TypeInfo(Double)) as TRttiFloatType;
WriteLn(Info.FloatType = ftSingle); //output: FALSE
WriteLn(Info.FloatType = ftDouble); //output: TRUE
```

- `TRttiSetType` adds an `ElementType` property returning the type information object for its elements. For example, the `TOpenOptions` set type of `System.UITypes` is defined as `set of TOpenOption`; in that case, `ElementType` will return a `TRttiType` object for `TOpenOption`. In the case of a set type not based on an explicit ordinal type, a valid a `TRttiType` object will still be returned.

- `TRttiStringType` adds a `StringKind` property. This is typed to `TRttiStringKind`, an enumeration with the possible values `skShortString`, `skAnsiString`, `skWideString` and `skUnicodeString`.

- `TRttiAnsiStringType` adds to `TRttiStringType` a `CodePage` property:

```
Info := Context.GetType(TypeInfo(UTF8String)) as TRttiAnsiStringType;
WriteLn(Info.CodePage); //output: 65001 (= CP_UTF8)
```

- `TRttiArrayType` is for static arrays only, and adds the following members:

```
property DimensionCount: Integer read GetDimensionCount;
property Dimensions[Index: Integer]: TRttiType read GetDimension;
property ElementType: TRttiType read GetElementType;
property TotalElementCount: Integer read GetTotalElementCount;
```

  The `Dimensions` property is such given a static array can have non-integral indexers:

```
type
  TSomeEnum = (Hello, Goodbye);
  TMyArray = array[Char, TSomeEnum] of Integer;
var
  Context: TRttiContext;
  ArrayType: TRttiArrayType;
  DimType: TRttiOrdinalType;
  I: Integer;
begin
  ArrayType := Context.GetType(TypeInfo(TMyArray)) as TRttiArrayType;
  WriteLn('No. of dimensions: ', ArrayType.DimensionCount);
  for I := 0 to ArrayType.DimensionCount - 1 do
  begin
    DimType := ArrayType.Dimensions[I] as TRttiOrdinalType;
    WriteLn('Type of dimension no. ', I + 1, ': ', DimType.Name);
    WriteLn('Ordinal low bound of dimension no. ', I + 1, ': ',
      DimType.MinValue);
      WriteLn('Ordinal high bound of dimension no. ', I + 1, ': ',
      DimType.MaxValue);
    end;
  WriteLn('Total element count: ', ArrayType.TotalElementCount);
end.
```

This outputs the following:

```
No. of dimensions: 2
Type of dimension no. 1: Char
Ordinal low bound of dimension no. 1: 0
Ordinal high bound of dimension no. 1: 65535
Type of dimension no. 2: TSomeEnum
Ordinal low bound of dimension no. 2: 0
Ordinal high bound of dimension no. 2: 1
Total element count: 131072
```

- `TRttiDynamicArrayType` exposes the following:

```
property DeclaringUnitName: string read GetDeclaringUnitName;
property ElementSize: Longint read GetElementSize;
property ElementType: TRttiType read GetElementType;
property OleAutoVarType: TVarType read GetOleAutoVarType;
```

  `OleAutoVarType` specifies the variant sub-type (i.e., `varXXX` value) that the element type would map to were an instance of the dynamic array converted to a variant array. This will be `$FFFF` (`65535`) in the case of an element type not variant array compatible.

- `TRttiPointerType` adds a `ReferredType` property. When the type being described is `Pointer` or a type alias for `Pointer` this returns `nil`, otherwise the type information object for the type pointed to is. For example, `ReferredType` for `PInteger` will return the `TRttiType` object for `Integer`.

- `TRttiInvokableType`, the base class for `TRttiMethodType` and `TRttiProcedureType`, introduces the following:

```
property CallingConvention: TCallConv read GetCallingConvention;
function GetParameters: TArray<TRttiParameter>;
function Invoke(const Callable: TValue;
  const Args: array of TValue): TValue; virtual;
property ReturnType: TRttiType read GetReturnType;
```

  `TCallConv` we’ve met before with `TRttiMethod` — it has the same meaning and status here. `TRttiInvokableType` also overrides `ToString` to return a string representation of the procedure or function’s declaration.

- `TRttiMethodType` adds to `TRttiInvokableType` a `MethodKind` property typed to `TMethodKind`, again something we’ve seen with `TRttiMethod`.

- `TRttiProcedureType` merely overrides inherited methods, and as such, does not introduce any further properties or public methods.

It may seem confusing at first that both `TRttiMethodType` and `TRttiMethod` classes are defined. The difference between them is that `TRttiMethodType` represents a method pointer type (e.g. `TNotifyEvent`, `TMouseEvent`, etc.) and `TRttiMethod` an actual method attached to a particular class, record or interface. Both nevertheless support calling methods dynamically.

Doing things with RTTI

The previous section having been an overview of the RTTI interface, in this one we will look at some examples of it being put to use.

Calling methods and procedures using TRttiMethodType and TRttiProcedureType

Either `TRttiMethod` or `TRttiMethodType` can be used to call an arbitrary method dynamically. The former will be the more usual one to use. For example, say you have an object with a method you want to call whose signature looks like the following:

```
function TryUpdateName(const Forename, Surname: string): Boolean;
```

Calling it dynamically using RTTI would look this this, assuming the method is on a class instance:

```
var
  Inst: TObject;
  InstType: TRttiInstanceType;
  Method: TRttiMethod;
  Success: Boolean;
begin
  //...
  Method := InstType.GetMethod('TryUpdateName');
  Success := Method.Invoke(Inst, ['John', 'Smith']).AsBoolean;
```

For a record, explicitly box the instance into a `TValue` before passing it as the first argument to `Invoke`.

In the case of `TRttiMethodType`, its `Invoke` method is passed not an instance of a class, record or interface, but a `TMethod` record that has been boxed into a `TValue`. `TMethod` is the internal representation of a method pointer:

```
type //defined in System
  TMethod = record
    Code, Data: Pointer;
  end;
```

Here, `Code` is the pointer to the procedure or function, and `Data` to the object whose method it is. Any variable, parameter or property with a method pointer type can be directly cast to a `TMethod`. This means `Invoke` may be called as so:

```
type
  TFooEvent = procedure (const S: string) of object;

procedure CallFooEventDynamically(const Handler: TFooEvent);
var
  Context: TRttiContext;
  EventType: TRttiMethodType;
begin
  EventType := Context.GetType(TypeInfo(TFooEvent)) as TRttiMethodType;
  EventType.Invoke(TValue.From(TMethod(Handler)), ['Blah']);
end;
```

Plainly, this example is contrived — in practice, you would just call `Handler` directly! Nonetheless, `CallFooEventDynamically` may be passed an actual method as thus:

```
type
  TTest = class
    procedure MyFooHandler(const S: string);
  end;

procedure TTest.MyFooHandler(const S: string);
begin
  WriteLn(S);
end;

var
  Obj: TTest;
begin
  Obj := TTest.Create;
  try
    CallFooEventDynamically(Obj.MyFooHandler);
  finally
    Obj.Free;
  end;
end.
```

In the case of `TRttiProcedureType`, its `Invoke` method is passed a simple pointer to a standalone procedure or function of the relevant signature, again boxed into a `TValue`. The compiler does need a little help to make the boxing work though:

```
uses
  System.Rtti;

type
  TFooFunc = function (const S: string): Boolean;

function CompatibleFunc(const S: string): Boolean;
begin
  WriteLn(S);
  Result := True;
end;

function CallStandaloneRoutineViaRTTI(AType: TRttiProcedureType;
  ARoutinePtr: Pointer; const Args: array of TValue): TValue;
var
  BoxedRoutinePtr: TValue;
begin
  TValue.Make(@ARoutinePtr, AType.Handle, BoxedRoutinePtr);
  Result := AType.Invoke(BoxedRoutinePtr, Args);
end;

var
  Context: TRttiContext;
  ProcType: TRttiProcedureType;
begin
  ProcType := Context.GetType(TypeInfo(TFooFunc)) as TRttiProcedureType;
  if CallStandaloneRoutineViaRTTI(ProcType, @CompatibleFunc,
    ['Can we do it?']).AsBoolean then WriteLn('Yes we can!');
end.
```

As this example shows, despite its name, `TRttiProcedureType` works with standalone functions as well as standalone procedures.

Creating objects

Given you can call an arbitrary method using RTTI, and constructors are a form of method, you can construct objects using RTTI. For example, say there is a class called `TFoo`, declared in the `FooStuff` unit, whose interface is as thus:

```
type
  TFoo = class
    constructor Create;
    procedure SayHello;
  end;
```

Creating an instance and calling the `SayHello` method using RTTI could be done like this:

```
uses System.Rtti;

var
  Context: TRttiContext;
  LType: TRttiInstanceType;
  Foo: TObject;
begin
  LType := Context.FindType('FooStuff') as TRttiInstanceType;
  Foo := LType.GetMethod('Create').Invoke(LType.MetaclassType,
    []).AsObject;
  try
    LType.GetMethod('SayHello').Invoke(Foo, []);
  finally
    Foo.Free;
  end;
end;
```

Notice we pass the metaclass for the ‘Self’ parameter for `Invoke`, which is what you must do for any constructor.

In principle, an object constructor need not be called `Create`. In the following helper function, we take account of this fact by looking for any public (or published) parameterless constructor, regardless of its name:

```
function CreateViaRtti(AType: TRttiInstanceType): TObject;
var
  Method: TRttiMethod;
begin
  for Method in AType.GetMethods do
    if (Method.MethodKind = mkConstructor) and
       (Method.Visibility >= mvPublic) and
       (Method.GetParameters = nil) then
    begin
```

```
      Result := Method.Invoke(AType.MetaclassType, []).AsType<T>;
      Exit;
    end;
  Assert(False);
end;
```

Given the public `Create` constructor defined by `TObject`, the assertion at the end should never be reached.

In practice, even doesn't make it completely robust though. In particular, what if the object to be constructed should be created using a constructor that takes one or more parameters? Examples include every single VCL or FireMonkey component, whose constructors all take an argument for their 'owner'.

Because of issues such as this, you should use `TRttiMethod` to construct objects only as a last resort, and in particular, when you can't use virtual constructors instead. In the case of the aforementioned components, this means casting the metaclass to `TComponentClass` and calling the `Create` method defined there:

```
function CreateComponent(AType: TRttiInstanceType;
  AOwner: TComponent): TComponent;
begin
  Assert(AType.MetaclassType.InheritsFrom(TComponent);
  Result := TComponentClass(AType.MetaclassType).Create(AOwner);
end;
```

Unlike in the RTTI case, this technique is statically typed, removing the possibility of runtime errors.

Cloning an arbitrary object

A step on from instantiating an arbitrary class with RTTI is cloning an arbitrary object. In the simpler cases, a method like the following will do the job — after constructing the new instance using the `CreateViaRtti` function previously listed, all public and published property values from a source object are copied over:

```
unit ObjectClone;

interface

type
  TObjectClone = record
    class function From<T: class>(Source: T): T; static;
  end;

implementation

uses
  System.SysUtils, System.Classes, System.TypInfo, System.RTTI;

class function TObjectClone.From<T>(Source: T): T;
var
  Context: TRttiContext;
  RttiType: TRttiInstanceType;
  Method: TRttiMethod;
  Prop: TRttiProperty;
begin
  RttiType := Context.GetType(Source.ClassType).AsInstance;
  Result := T(CreateViaRtti(RttiType));
  try
    for Prop in RttiType.GetProperties do
      if (Prop.Visibility >= mvPublic) and Prop.IsReadable and
         Prop.IsWritable then
        Prop.SetValue(TObject(Result), Prop.GetValue(TObject(Source)));
  except
    Result.Free;
    raise;
  end;
end;

end.
```

In use, it works like this:

```
procedure CreateAndUseClone(SourceObj: TMyObject);
var
  Doppelgänger: TMyObject;
begin
  Doppelgänger := TObjectClone.From(SourceObj);
  //work with Doppelgänger...
```

Nonetheless, the caveats about instantiating an arbitrary class using RTTI apply even more to cloning an arbitrary object. For example, some classes have properties that provide a different view to the same data (e.g. the `CommaText`, `DelimitedText` and `Text` properties of `TStrings`), some care about the order properties are set (e.g. the `Parent` property of a VCL control usually needs to be set before anything else), and some have built-in support for cloning that should surely be used instead of a generalised RTTI-based approach (e.g. `TPersistent` descendants may implement either `Assign` or `AssignTo`). Another area of possible complexity is how certain properties may need special handling, for example a property typed to a class may require its properties to be copied over manually.

Finding all descendant classes of a given base class

While robustly cloning objects using RTTI is a task that suffers from hidden complexities, enumerating all descendants of a given class is quite a simple affair. The following example finds all `TStrings` descendants:

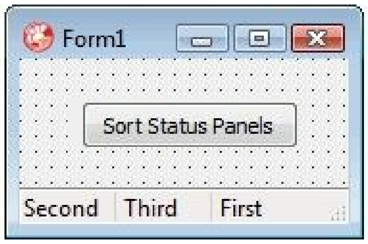
```
uses
  System.SysUtils, System.Classes, System.Rtti;

var
  Context: TRttiContext;
  LType: TRttiType;
  Found: TClass;
begin
  for LType in Context.GetTypes do
    if LType.IsInstance then
    begin
      Found := TRttiInstanceType(LType).MetaclassType;
      if Found.InheritsFrom(TStrings) and (Found <> TStrings) then
        WriteLn(Found.ClassName);
    end;
end.
```

Despite its name, `InheritsFrom` returns `True` if the class equals the one queried for, which is why we check for `TStrings` itself here.

Accessing private fields

In an ideal world, a class will expose all that is necessary to use it efficiently. In practice, this isn’t always the case, which can cause difficulties if the class is one you cannot change. For example, the `TCollection` class, which implements a streamable list and is used throughout the VCL, does not expose a `Sort` method. As a result, sorting a collection requires items to be repeatedly removed and added back in. Nonetheless, a `TCollection` instance is based around an internal `TList` that *does* provide sorting functionality. Using RTTI, this `TList` can be accessed directly. Doing such a thing is ultimately a hack — encapsulation is generally a Good Thing, and should be broken only with good reason. Nonetheless, the fact the default RTTI settings cause type information to be generated for private fields means the practicalities of breaking it in this particular case aren’t very taxing at all.

To see this in action, create a new VCL application, and add a `TStatusBar` and a `TButton` to the form. Add a few panels to the status bar so that the form looks like this at design time:



We will now handle the button’s `OnClick` event to sort the status panels (i.e., the things captioned ‘Third’, ‘Second’ and ‘First’). To do this, add the following code to the top of the `implementation` section of the unit, immediately below the `{$R *.dfm}`:

```
uses System.Rtti;

type
  TCollectionAccess = class(TCollection);

procedure SortCollection(ACollection: TCollection);
var
  Context: TRttiContext;
  InstType: TRttiInstanceType;
  List: TList;
```

```
begin
  InstType := Context.GetType(ACollection.ClassType).AsInstance;
  List := InstType.GetField('FItems').GetValue(
    ACollection).AsType<TList>;
  List.SortList(
    function (Item1, Item2: Pointer): Integer
    begin
      Result := CompareText(TCollectionItem(Item1).DisplayName,
        TCollectionItem(Item2).DisplayName, loUserLocale);
    end);
  TCollectionAccess(ACollection).Changed;
end;
```

Here, we first get the type information object for the collection, before retrieving the value of its (private) `FItems` field. This should be typed to the non-generic `TList`, whose `SortList` method may then be called directly. Finally, the collection is told changes in its items have happened, which will have the effect of updating the display as necessary. Helper method written, the button's `OnClick` event can be handled like so:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SortCollection(StatusBar1.Panels);
end;
```

If you run the application, you should find clicking the button does indeed sort the status panels.

RTTI limitations

Before moving on, it is important to note a few limitations of 'extended' RTTI:

- Both open array and untyped parameters are not supported. When a method has one of these, the method itself will have a retrievable `TRttiMethod` object, however `HasExtendedInfo` will return `False`, and you won't be able to enumerate the described method's parameters.
- RTTI generation for interface types is almost non-existent by default. To fix this, each base interface type needs to be compiled with the `$M+` switch enabled:

```
  IMyInterfaceWithLittleRTTI = interface
  ['{59EC4589-3AEA-1069-A2DD-08002B30309D}']
    procedure Foo;
  end;

  {$M+}
  IMyInterfaceWithNormalRTTI = interface
  ['{4AEC4589-3AEA-1069-A234-08002B30309D}']
    procedure Foo;
  end;
```

- Generic types in their pre-instantiated state do not exist at all from the vantage of the RTTI system. Thus, if you use `TList<Integer>` somewhere, then the `TList<Integer>` class will be discoverable from a `TRttiContext` record; however, `TList<T>` will not be.
- Anonymous method types are exposed as interface types with a single method, `Invoke`. As with interfaces generally, you need to use the `$M+` switch for proper RTTI to be generated:

```
  uses
    System.Rtti;

  type
    {$M+}
    TAnonProc = reference to procedure (const S: string);

  var
    Context: TRttiContext;
    ProcType: TRttiType;
    Method: TRttiMethod;
  begin
    ProcType := Context.GetType(TypeInfo(TAnonProc));
    WriteLn(ProcType.ClassName);
    for Method in ProcType.GetMethods do
      WriteLn(Method.ToString);
  end.
```

  This program outputs `TRttiInterfaceType` for the class name followed by `procedure Invoke(const S: string)` for the single method.

Adding custom RTTI: using attributes

An 'attribute' in Delphi is a piece of metadata, attached in code to a type or individual method, that can be queried for at runtime. Conceptually, Delphi attributes are very similar to attributes in .NET and annotations in Java. Nonetheless, Delphi attributes don't play quite the role their equivalents do in those other languages. This is because ordinary compiler directives take care of certain basic metadata needs.

For example, say you wish to mark a unit, method or type deprecated (obsolete) so that the compiler issues a warning when it gets used. Where in .NET this would be done through a system-defined attribute, and in Java a system-defined annotation, in Delphi you use the `deprecated` directive:

```
type
  TNeanderthal = class
    procedure Foo;
  end deprecated 'Class superseded by THomoSapiens';

  THomoSapiens = class
    procedure OldProc; deprecated 'Use NewProc instead';
    procedure NewProc;
  end;
```

Similarly, a unit, method or type may also be marked platform specific with the `platform` directive, and experimental with the `experimental` directive:

```
type
  TWinSpecificObject = class
    class procedure BrandNewProc; experimental;
    class procedure TriedAndTestedProc;
  end platform;
```

Another role that attributes could have in principle but don't have in practice is suppressing compiler warnings. Instead, a choice of two comment-style directives are available, `$WARNINGS` and `$WARN`:

```
{$WARN SYMBOL_PLATFORM OFF} //disable platform-specific warnings
procedure DodgyAndItKnowsItCode(Obj: TEvolvingObject);
begin
  TWinSpecificObject.TriedAndTestedProc;
  {$WARNINGS OFF}               //disable *all* warnings
  ExtremelyBadCode(UnicodeString(PAnsiChar(@Obj)));
  {$WARNINGS ON}                //re-enable them
  {$WARN SYMBOL_DEPRECATED OFF} //disable  'deprecated' warnings
  Obj.OldProc;
  {$WARN SYMBOL_DEPRECATED ON}  //re-enable them
end;
{$WARN SYMBOL_PLATFORM ON}      //re-enable platform warnings
```

Nonetheless, use of attributes is slowly creeping in to Delphi's standard libraries. For example, a custom component must use an attribute to tell the IDE what platforms it supports:

```
[ComponentPlatforms(pidWin32 or pidWin64 or pidOSX32)]
TMyAllRoundComponent = class(TComponent)
//...
end;
```

Defining and applying attribute types

In terms of actual coding, an attribute is defined by a class that descends from `TCustomAttribute`. If the attribute has no parameters, it can descend from `TCustomAttribute` without adding anything:

```
type
  WellTestedAttribute = class(TCustomAttribute);
```

Once a custom attribute is defined, it can be declared an attribute of specific classes, fields, properties or methods. This is done by entering the attribute's name in square brackets immediately before the thing it is being made an attribute of. As a convenience, if the attribute class name ends in the word `Attribute`, that word can be dropped:

```
type
  [WellTestedAttribute] //OK
  TExample = class
    [WellTested]        //also OK
    procedure Foo;
  end;
```

In this case, both `TExample` as a whole and its `Foo` method specifically are being marked as 'well tested'.

(As an aside, while you can drop any '-Attribute' suffix when applying an attribute class, you cannot do the same with any T prefix. Because of this, attribute types constitute one case where the normal convention of prefixing class names with a T is not usually followed.)

If you so wish, more than one attribute class can be applied to the same target, each being declared within its own square bracket pair:

```
type
  [WellTested]
  [ComponentPlatforms(pidWin32 or pidOSX32)]
  TAnotherExample = class(TComponent)
    //...
  end;
```

Attributes with parameters

Any attribute can be coded to take one or more parameters. To do this, provide the attribute class with a constructor. This usually does no more than assign fields, which are themselves straightforwardly exposed as public properties. For example, the following is an attribute that takes two string parameters:

```
type
  AppraisalAttribute = class(TCustomAttribute)
  strict private
    FComment, FName: string;
  public
    constructor Create(const AComment, AName: string);
    property Comment: string read FComment;
    property Name: string read FName;
  end;

constructor AppraisalAttribute.Create(const AComment, AName: string);
begin
  inherited Create;
  FComment := AComment;
  FName := AName;
end;
```

When applied, an attribute's parameters are listed in brackets immediately after the type name, separated with commas like you would separate the arguments of a procedure or function:

```
type
  [Appraisal('This class needs to be more fleshed out',
    'Joe Bloggs')]
  TExample = class
    [Appraisal('Well tested', 'John Smith')]
    procedure Foo;
  end;
```

All arguments must be included, and in the order they appear in the constructor's header. If one or more should be merely optional, either overload the constructor or use default parameters, as you might for any other routine:

```
type
  AppraisalAttribute = class(TCustomAttribute)
  //rest as before...
    constructor Create(const AComment: string;
      const AName: string = '');
  end;

type
  [Appraisal('This comment needs to be fleshed out more')]
  TYetAnotherExample = class
    //...
  end;
```

Querying for attributes

At runtime, find out what attributes a type or method has via a `TRttiContext` record, enumerating the `TCustomAttribute` array returned by the `GetAttributes` method of the relevant `TRttiType` instance:

```
procedure WriteAppraisalInfo(AClass: TClass);
var
  Context: TRttiContext;
  LType: TRttiType;
  Attrib: TCustomAttribute;
  Appraisal: AppraisalAttribute;
  Method: TRttiMethod;
```

```
begin
  LType := Context.GetType(AClass);
  if LType = nil then Exit;
  for Attrib in LType.GetAttributes do
    if Attrib is AppraisalAttribute then
    begin
      Appraisal := AppraisalAttribute(Attrib);
      WriteLn(AClass.ClassName, ': ',
        Appraisal.Comment, ' [', Appraisal.Name, ']');
      Break;
    end;
  for Method in LType.GetMethods do
    for Attrib in Method.GetAttributes do
      if Attrib is AppraisalAttribute then
      begin
        Appraisal := AppraisalAttribute(Attrib);
        WriteLn('  Method ', Method.Name, ': ',
          Appraisal.Comment, ' [', Appraisal.Name, ']');
        Break;
      end;
end;

begin
  WriteAppraisalInfo(TExample);
end.
```

Assuming `TExample` was declared as above, this program outputs the following:

```
TExample: This class needs to be more fleshed out [Joe Bloggs]
  Method Foo: Well tested [John Smith]
```

Attribute usage scenarios

Even more than RTTI in general, attributes are a 'framework-y' feature. In other words, while application-level code is unlikely to find much utility in the ability to define custom attribute classes, framework writers can find attributes a very important feature. A good example of this is DSharp, an open source library by Stefan Glienke (`http://code.google.com/p/dsharp/`). Amongst various things, this project implements an aspect-oriented programming (AOP) framework, a data binding system for controls, and a generic way to serialise objects to XML — and all three leverage attributes.

In the first case, attributes are used to tag particular classes or methods as desiring certain functionality provided by an 'aspect'. As an example, the framework provides a logging aspect, which writes details of method calls to a log of some sort. In use, you simply mark a class or method with the `Logging` attribute defined by the framework; taking no parameters when applied, this and other aspect attributes specify to the framework a class that implements the 'aspect':

```
{ DSharp.Aspects.pas }

type
  TAspect = class
  public
    //... (various virtual class methods)
  end;

  TAspectClass = class of TAspect;

  AspectAttribute = class(TCustomAttribute)
  private
    FAspectClass: TAspectClass;
  public
    constructor Create(AspectClass: TAspectClass);
    property AspectClass: TAspectClass read FAspectClass;
  end;

constructor AspectAttribute.Create(AspectClass: TAspectClass);
begin
  FAspectClass := AspectClass;
end;

{ DSharp.Aspects.Logging.pas }

type
  TLoggingAspect = class(TAspect)
  public
    //... (implements methods introduced in abstract base class)
  end;
  LoggingAttribute = class(AspectAttribute)
  public
    constructor Create;
  end;

constructor LoggingAttribute.Create;
begin
  inherited Create(TLoggingAspect);
end;
```

At runtime, the aspects engine then uses `TRttiContext` to cycle through the application's types and methods, looking for items marked with an aspect attribute. When it finds one, the aspect implementation is picked off and set up for the type or method concerned.

The way DSharp uses attributes in the data binding and XML cases is similar if not identical. In the former, a `BindingAttribute` is defined to allow specifying binding details for individual fields (in particular, their source and target). In the latter, custom attributes are provided to allow overriding the default serialisation behaviour: where one attribute allows specifying that a certain member should *not* be serialised, another allows saying it should be serialised to a specific element, as opposed to just an element named after the field or method itself:

```
type
  TSomeObj = class
  published
    [XmlIgnoreAttribute]
    property NonSerializedData: string read FNonSerializedData
      write FNonSerializedData;
    [XmlElementAttribute('Foo')]
    property Bar: string read FBar write FBar;
  end;
```

13. Working with native APIs

The coverage of Delphi’s standard libraries is wide enough for most tasks, and moreover, using them usually brings the benefit of allowing the same code to be employed across different platforms. However, there may on occasion be the need to drop down a level and use the native API (Application Programming Interface) directly. This chapter looks at how to go about doing that.

We will begin with the preliminary issue of ‘conditional compilation’, which allows marking certain areas of code as specific to a specific platform, before turning to the syntax for using the native APIs themselves. In the case of Windows this will include looking at the language side of Delphi’s COM (Component Object Model) support; in the case of OS X, it will involve how to work with the ‘Cocoa’ API layer using XE2’s Delphi to Objective-C bridge.

Using conditional defines

When developing for different platforms, the chances are certain pieces of code will be specific to one platform and others specific to a second. In order to avoid having to mess about swapping different units in and out, you will need to use ‘conditional compilation’. This allows marking a given area of code as compilable only if a certain compile-time condition or set of conditions is met — if the condition isn’t met, then the code is effectively commented out.

To use conditional compilation, wrap code in either `$IFDEF`/`$ENDIF` or `$IF`/`$IFEND` blocks. For example, Windows-specific code can be marked like this:

```
{$IFDEF MSWINDOWS}
  //do something Windows-specific
{$ENDIF}
```

Aside from `MSWINDOWS`, other standard ‘conditional defines’ include `WIN32`, `WIN64`, `MACOS` and `POSIX` — in XE2, `POSIX` is effectively a synonym for `MACOS`, however in previous and possibly later versions it was defined for Linux too. It is also possible to create custom conditional defines; this you can do either in code — use the `{$DEFINE MYCUSTOMDEFINE}` syntax — or in the IDE under Project Options. Once created, a conditional define can also be undefined using `$UNDEF`:

```
{$DEFINE MYDEFINE} //define MYDEFINE

{$UNDEF MYDEFINE}  //MYDEFINE is no longer defined
```

Making use of `$UNDEF` can prove confusing however.

$IF vs. $IFDEF

Compared to `$IFDEF`, the `$IF` syntax is a bit more flexible because it allows testing for the existence of certain symbols (e.g. whether a given type is in scope) as well conditional defines: to test for a symbol use `$IF DECLARED(SymbolName)`, and for a conditional define use `$IF DEFINED(CondDefineName)`. In fact, using `$IF`, you can conditionally compile relative to the result of any `Boolean` constant expression:

```
{$IF DEFINED(MSWINDOWS)}
  //do something Windows-specific
{$ELSEIF DEFINED(MACOS)}
  //do something Mac-specific
{$ELSE}
  {$MESSAGE WARN 'Only tested on Windows and the Mac'}
{$IFEND}

{$IF SomeConst = 42}
  //compile only if SomeConst equals 42
{$IFEND}

{$IF NOT DECLARED(UnicodeString)}
type
  UnicodeString = WideString; //Backfill UnicodeString type when
{$IFEND}                      //compiling in old Delphi version
```

In all cases, use of `$ELSEIF` add/or `$ELSE` is optional.

Using the Windows API

The classic Windows API is implemented across several DLLs (dynamic link libraries) and is designed to be callable from programs written in 'C'. As a programming language, C is relatively simple when you look at the breadth of features it supports, but relatively tricky to actually program in. The API reflects both aspects.

Being just normal DLLs, system libraries contain very little metadata describing what functions they export — for any given DLL, it is possible to enumerate the names of exported functions (the free Dependency Walker utility — depends.exe — is excellent for this), but parameter types, result types and (when compiling for 32 bit) the 'calling convention' need to be known independently. However, since the API is well documented by Microsoft, and Delphi comes with a comprehensive set of 'import units' or translations of the original C interfaces into Pascal, that isn't really a problem in practice.

These import units are equivalent to the header files you would use if programming against the Windows API in C or C++. Nonetheless, one immediate difference is that the most commonly-used header files in a C or C++ context are combined into a single Delphi import unit, `Winapi.Windows`. This contains declarations for the 'kernel', 'GDI' and 'Registry' APIs, amongst others. The fact they are grouped together is purely to avoid clogging up the typical uses clause; for the same reason, the core COM types and constants are grouped together in the `Winapi.ActiveX` unit. Beyond these exceptions, the names of the stock import units correspond to the C header file names though.

For example, if you wish to programmatically open a file as if it were double clicked in Windows Explorer, the API function to call is `ShellExecute`. In a C or C++ context you would import the `Shellapi.h` header file to access it; in Delphi, you use the `Winapi.Shellapi` unit:

```
uses Winapi.ShellApi;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShellExecute(0, nil, 'C:\Users\CCR\Documents\Notes.txt',
    nil, nil, SW_SHOWNORMAL)
end;
```

Here, `NULL` in C becomes `nil` in Delphi, but other than that, the call looks much the same made in either language.

When calling the Windows API, provided type aliases should be used as much as possible, in the same way a C or C++ programmer should use the provided typedefs. If the documentation on MSDN or the Platform SDK says a given function returns an `HBITMAP`, for instance, put the result into a variable typed to `HBITMAP`, not `LongWord` or whatever. Doing this is particularly important if you wish to cross compile between 32 bit and 64 bit targets, since the mapping of an API type can change between Win32 and Win64. For example, all the 'handle' types (`HBITMAP`, `HDC`, `HWND`, etc.) are 32 bit quantities when targeting Win32, but 64 bit ones when targeting Win64.

The one exception to the 'use provided type aliases' rule is `INT` — due to a name clash with the `Int` standard function, you must just use `Integer` instead. The `HANDLE` type also doesn't exist, but in that case it is just renamed — use `THandle`. Similarly, all 'struct' (i.e., record) types are given more natural-looking aliases too, though in their case the 'real' `ALLCAPS` names are defined as well. Thus, the `GetIconInfo` API function, which takes an `ICONINFO` struct according to MSDN, takes either an `ICONINFO` or `TIconInfo` record in Delphi (the two types are interchangeable):

```
uses System.SysUtils, Winapi.Windows, Vcl.Graphics;

procedure ExtractIconBitmaps(AIcon: TIcon; AImage, AMask: TBitmap);
var
  Info: TIconInfo;
begin
  if not GetIconInfo(AIcon.Handle, Info) then
    RaiseLastOSError; //turn an OS error into a Delphi exception
  AImage.Handle := Info.hbmColor;
  AMask.Handle := Info.hbmMask;
end;
```

Sending strings to a Windows API function

A common requirement when calling the API is to pass string data. Due to its C origins, the classic Windows API therefore uses C-style 'strings', which in Delphi terms means the `PChar`/`PWideChar` and `PAnsiChar` types we looked at the end of chapter 5.

Before Windows NT came on the scene, the API assumed text was encoded in the system 'Ansi' encoding; with Windows NT (which eventually became XP, Vista, 7 and so forth), the API was 'doubled up' to include parallel Unicode functions. Internally, the legacy 'Ansi' functions were given an 'A' suffix (e.g. `GetWindowTextA`) and the more modern Unicode ('wide') functions a 'W' suffix (e.g. `GetWindowTextA`). If programming in C++, a compiler directive controls what variant the generic, non-suffixed API identifiers (e.g. `GetWindowText`) are mapped to. In Delphi, this is however fixed to the `xxxW` variants. This is consistent with the mapping of the `string` type to `UnicodeString`, and in practical terms, allows 'converting' to a C-style 'string' appropriate for the API with a simple typecast:

```
uses Winapi.Windows;

procedure SetEnvironmentVar(const AName, AValue: string);
begin
  SetEnvironmentVariable(PChar(AName), PChar(AValue));
end;
```

Historically, `string` in Delphi mapped to `AnsiString`; when it did, the generic API functions mapped to the `xxxA` variants. The fact both mappings changed in step meant old code still worked.

Some newer API functions only come in a W form, in which case `string` being `UnicodeString` simplifies matters compared to times past. However, a few older functions only have an 'A' form, for example `WinExec`. Most (possibly all) have long been deprecated by Microsoft — in the case of `WinExec`, for example, you are supposed to use the rather more complicated `CreateProcess` function. If you must call `WinExec` though, pass a native string with a double cast, first to `AnsiString` and then on to `PAnsiChar`:

```
uses
  Winapi.Windows;

procedure DoWinExec(const ACommandLine: string; ACmdShow: UINT);
begin
  WinExec(PAnsiChar(AnsiString(ACommandLine)), ACmdShow);
end;
```

This advice holds if passing a string literal or true constant, though in those cases, the compiler will make no objection to them being passed directly:

```
const WordPad = 'wordpad.exe';         //true constant

WinExec('notepad.exe', SW_SHOWNORMAL); //use literal directly
WinExec(WordPad, SW_SHOWNORMAL);      //use constant directly
```

Given `WinExec` is declared to expect a `PAnsiChar`, the compiler will ensure an 'Ansi'-encoded C-style string is passed to it in both cases.

Receiving strings from Windows API functions

In the examples so far, we've passed *in* a string to the Windows API. What about receiving one? This is slightly trickier, since a C-style 'string' implies nothing of how its memory was allocated — it's just a pointer to an array of `AnsiChar` or `WideChar` values that ends in a zero (the 'null terminator'), no more no less. When it needs to return a string, a classic Windows API function therefore usually requests two things, a block of allocated memory (the 'buffer') to copy into and the maximum number of characters it can copy (the size of the buffer in other words). On return, the buffer is filled, and the number of characters copied returned as the function result.

In Delphi, the buffer can be either a Delphi string initialised with a `SetLength` call or a local static array of `Char`, which can then be assigned to a string. For example, say you have a handle to an API-level window that you want to get the text or caption for. In that case, you would be wanting to call the `GetWindowText` API function. Using the `SetLength` approach goes like this:

```
uses Winapi.Windows;

var
  Text: string;
  Wnd: HWND;
begin
  //... assign Wnd from wherever
  SetString(Text, GetWindowTextLength(Wnd));
  GetWindowText(Wnd, PChar(Text), Length(Text));
  //... work with Text
end;
```

Alternatively, a static array could be used like this, assuming you don't mind having a fixed cap on the number of characters returned:

```
uses Winapi.Windows;

var
```

```
  Buffer: array[0..1023] of Char;
  Len: Integer;
  S: string;
  Wnd: HWND;
begin
  //... assign Wnd from wherever
  Len := GetWindowText(Wnd, Buffer, Length(Buffer));
  SetLength(Text, Buffer, Len);
  //... work with Text
end;
```

The usual case is for the Windows API to require buffer sizes specified in *elements* not bytes (if the documentation on MSDN talks in terms of the number of '`TCHAR` values', then the number of elements is needed). Consequently, you should be careful to use `Length(Buffer)` not `SizeOf(Buffer)`, since in the `array of Char` case, `SizeOf` will return a figure twice as big as `Length`.

Calling conventions on Windows

A 'calling convention' defines the precise way in which a compiler implements how a procedure or function is called. When creating 32 bit programs or libraries, the Delphi compiler uses its own convention by default ('register', sometimes called 'fastcall' or 'Borland fastcall') which differs from the one used by the Windows API ('stdcall'). Since import units for the API have been written for you, this generally isn't an issue. Nonetheless, it will be if you call an API routine that takes a 'callback', i.e. procedure or function that the API routine itself then calls. In that case, you must be careful to add `stdcall;` to the end of the callback's declaration:

```
function MyCallback(P1: SomeType; ...): BOOL; stdcall;
```

For example, say you want to enumerate the top-level windows currently running on the system. For this task, the `EnumWindows` API routine is ideal. In use, it is passed the address of a function, which it then calls repeatedly for each top-level window it finds:

```
uses Winapi.Windows;

function ProcessWnd(Wnd: HWND; lParam: LPARAM): BOOL; stdcall;
var
  S: string;
begin
  SetLength(S, GetWindowTextLength(Wnd));
  SetLength(S, GetWindowText(Wnd, PChar(S), Length(S));
  WriteLn(WindowTitle);
  Result := True;
end;

begin
  WriteLn('The captions of top level windows are as thus: ');
  EnumWindows(@ProcessWnd, 0);
end.
```

Interoperability with user interface APIs

When creating a Windows GUI application in Delphi, there is a choice between two frameworks, the Visual Component Library (VCL) and FireMonkey (FMX). Despite being the older platform, the VCL has various things in its favour if you don't have any interest in FMX's cross platform support. Making a proper comparison is outside the scope of this book, however one important aspect is the fact the VCL has much better interoperability with the native API.

This is not to say you cannot call the API when writing a FireMonkey application — the `Winapi.Windows` unit and so forth are still available, and any API function not directly related to the user interface can be called as normal. However, the VCL is fundamentally more API-friendly because at heart, it is a wrapper round the native user interface APIs: a VCL `TButton` wraps an API-implemented button control, a VCL `TTreeView` wraps an API-implemented tree view control, and so on.

Because of this origin, many VCL classes have a `Handle` property that allows direct access to an underlying API primitive:

- Forms and 'windowed' controls (i.e., `TWinControl` descendants, e.g. `TButton`, `TListView`, etc.) have a `Handle` property typed to `HWND`.
- The VCL `TBitmap` has a `Handle` property typed to `HBITMAP`.
- The VCL `TBrush` has a `Handle` property typed to `HBRUSH`.
- `TIcon` has a `Handle` property typed to `HICON`.

- `TImageList` has a `Handle` property typed to `HIMAGELIST`.
- `TMetafile` has a `Handle` property typed to `HMETAFILE`.
- The VCL `TCanvas` (which wraps a 'device context' — DC) has a `Handle` property typed to `HDC`.

These properties have various benefits, the first being how they allow calling API functions against VCL objects to fill functionality gaps in the VCL itself.

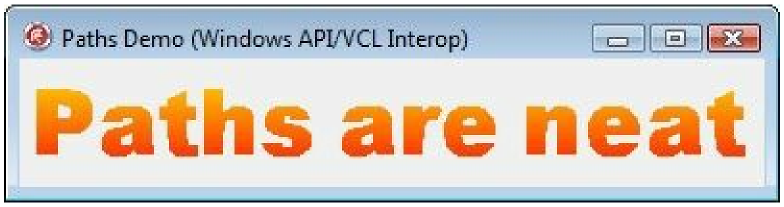
For example, the VCL `TCanvas` does not have a method for drawing text with a gradient fill. Nonetheless, the `Vcl.GraphUtil` unit has a routine, `GradientFillCanvas`, for filling out a rectangular area with a gradient, and at the Windows API level, 'path' functionality allows creating a 'clipping region' based on the shape of a piece of text (when a 'clipping region' is set, drawing operations will only output to the space within it). Put these two things together, and a routine for drawing gradient text can be written like this:

```
uses Vcl.GraphUtil;

procedure DrawGradientText(ACanvas: TCanvas; const AText: string;
  X, Y: Integer; AStartColor, AEndColor: TColor;
  ADirection: TGradientDirection);
var
  DC: HDC;
  R: TRect;
  SaveIndex: Integer;
begin
  { Drop down to the Windows API to save the state of the
    canvas }
  DC := ACanvas.Handle;
  SaveIndex := SaveDC(DC);
  try
    { Compose the destination rectangle }
    R.Left := X;
    R.Top := Y;
    R.Right := X + ACanvas.TextWidth(AText);
    R.Bottom := Y + ACanvas.TextHeight(AText);
    { Using a mixture of direct API and VCL calls, create the
      path, select it as the clipping region, and fill it with
      the gradient }
    BeginPath(DC);
    ACanvas.TextOut(R.Left, R.Top, AText);
    EndPath(DC);
    SelectClipPath(DC, RGN_DIFF);
    GradientFillCanvas(ACanvas, AStartColor, AEndColor,
      R, ADirection);
  finally
    { Drop down to the API once more to restore the original
      state of the canvas }
    RestoreDC(DC, SaveIndex)
  end;
end;
```

To see this in action, create a new VCL Forms application and add the code just listed to the top of the unit's `implementation` section. Then, handle the form's `OnPaint` event as thus:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  Size: TSize;
  TextToDraw: string;
begin
  TextToDraw := 'Paths are pretty neat';
  Canvas.Font.Name := 'Arial Black';
  Canvas.Font.Size := 36;
  Size := Canvas.TextExtent(TextToDraw);
  DrawGradientText(Canvas, TextToDraw, (ClientWidth - Size.cx) div 2,
    (ClientHeight - Size.cy) div 2, clYellow, clRed, gdVertical);
end;
```

This results in output like the following:



Writing the Handle property

In the case of `TBitmap`, `TBrush`, `TIcon` and `TImageList`, along with `TCanvas` objects not attached to a form or control, the `Handle` property is writeable. This allows you to initialise certain VCL objects from an API routine. For example, a resource icon could be loaded into a `TIcon` at a specific size by calling the `LoadImage` API function:

```
{ Load the application's main icon (set in the IDE via
  Project|Options, Application) at 64x64 }
Icon.Handle := LoadImage(HInstance, 'MAINICON', IMAGE_ICON,
  64, 64, LR_DEFAULTCOLOR);
```

Another usage scenario of writeable `Handle` properties is when you have an API primitive to work with, but would prefer to use the VCL equivalent given it is easier to use. For example, to customise the painting of a form's background, you might handle the `WM_ERASEBKGND` message. In this case, the system will provide you with an `HDC`. To work with it through a `TCanvas`, just create a temporary instance and assign the `Handle` property to the `HDC` you were given:

```
var
  Canvas: TCanvas;
begin
  Canvas := TCanvas.Create;
  try
    //assign the handle
    Canvas.Handle := Message.DC;
    //draw to the DC using TCanvas members
    Canvas.Brush.Color := clLtGray;
    Canvas.FillRect(Rect(5, 5, 100, 100));
    Canvas.Draw(10, 10, Image1.Picture.Graphic);
    //...
  finally
    Canvas.Free;
  end;
```

In all but the `TCanvas` case, the API-level handle is 'owned' by the VCL object. This means after assigning the `Handle` property, there is no need to call `DeleteObject` or equivalent after you have finished, as you would normally need to do if writing purely API-level code that works with bitmaps and brushes and so forth.

This behaviour may occasionally be undesirable however. For example, there may be times when you want to use the VCL to create an API-level primitive that will be passed to an API function which is documented to own the handle itself. In such a situation, you just need to call the VCL object's `ReleaseHandle` method. Usually this is done immediately before freeing the VCL object:

```
function LoadHBITMAPFromFile(const AFileName: string): HBITMAP;
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile(AFileName);
    Result := Bitmap.ReleaseHandle;
  finally
    Bitmap.Free;
  end;
end;
```

If `ReleaseHandle` weren't called, the `HBITMAP` would be deleted by the `TBitmap` destructor.

Windows messaging

Aside from the `Handle` properties, the second major interoperability feature of the VCL is integrated support for window 'messaging'. 'Sending' or 'posting' a message is the API equivalent of making a method call, or reading or writing a property. For example, say you have a `TEdit` control and set its `Text` property. Since `TEdit` wraps a native edit control, the `Text` property setter is implemented by sending a `WM_SETTEXT` message to the underlying API primitive.

If writing a custom control based on `TEdit` (or `TCustomEdit`, its immediate ancestor), the normal processing of individual messages can be intercepted using the message handling syntax we met in chapter 3:

```
type
  TMyEdit = class(TEdit)
  protected
    procedure WMSetText(var Msg: TWMSetText); message WM_SETTEXT;
  end;

procedure TMyEdit.WMSetText(var Msg: TWMSetText);
```

```
begin
  //when debugging, send text being set to IDE's event log
  if DebugHook <> 0 then
    OutputDebugString(Message.Text);
  //get the text actually set!
  inherited;
end;
```

While the name of the method can be anything, usual practice is to name it after the message, as shown here (i.e., `WMSuchAndSo` for `WM_SUCHANDSO`). Most standard messages have record types declared for them in the `Winapi.Messages` unit; if one isn't, you can just use `TMessage`:

```
procedure WMUnusual(var Msg: TMessage); message WM_UNUSUAL;
```

When it comes to sending or posting a message in the first place, the API functions to call are `SendMessage` and `PostMessage`, as declared in `Winapi.Windows`:

```
SendMessage(SomeControl.Handle, WM_FOO, 0, 0);
PostMessage(SomeControl.Handle, WM_FOO, 0, 0);
```

The difference between the two is that sending a message waits for the receiver to respond where posting doesn't (consequently, not all messages support posting — `WM_SETTEXT` is an example of a send-only message). If sending to a VCL control, an alternative to `SendMessage` is calling the `Perform` method:

```
SomeControl.Perform(WM_SETTEXT, 0, LPARAM(PChar('Blah')));
```

Using `Perform`, the message is sent directly to the control rather than via the application's message loop.

The various parts of Delphi's messaging support work with forms too. For example, when running on battery power, the system will post a `WM_POWERBROADCAST` message to all top level windows when the power status changes. Since the main form of a VCL application is a top level window in API terms, it will receive `WM_POWERBROADCAST` messages too. These can then be handled in the normal way:

```
type
  TMyForm = class(TForm)
  //...
  protected
    procedure WMPowerBroadcast(var Message: TMessage);
      message WM_POWERBROADCAST;
  //...
  end;

procedure TMyForm.WMPowerBroadcast(var Message: TMessage);
var
  Status: TSystemPowerStatus;
begin
  //see MSDN for details about WM_POWERBROADCAST
  case Message.WParam of
    PBT_APMBATTERYLOW: Caption := 'Battery is low!';
    PBT_APMPOWERSTATUSCHANGE:
      if GetSystemPowerStatus(Status) then
        case Status.BatteryFlag of
          2: Caption := 'Battery is low!';
          4: Caption := 'Battery is critically low!';
        end;
  end;
end;
```

FMX interoperability

Describing FireMonkey's interoperability features is easy, since they amount to a pair of functions to convert between a reference to an FMX form and an `HWND`:

```
function FindWindow(Handle: HWND): TCommonCustomForm;
function FmxHandleToHWND(FmxHandle: TFmxHandle): HWND;
```

These are both declared by the `FMX.Platform.Win` unit; on failure `FindWindow` returns `nil` where `FmxHandleToHWND` raises an `EInvalidFmxHandle` exception. Call it by passing the `Handle` property of a FMX form:

```
procedure TMyForm.btnDoSomethingClick(Sender: TObject);
var
  Wnd: HWND;
begin
  Wnd := FmxHandleToHWND(Handle);
  //... use Wnd
```

The VCL equivalents are the `FindControl` function in `Vcl.Controls` (this returns a `TWinControl`, which may or may not be a form) and the `Handle` property, which on either a form or control is an `HWND` directly.

Given the scope of FMX is much smaller than the VCL, the lack of 'hooks' to the native API can be problematic. Pragmatically, the VCL can therefore prove a useful interoperability tool in itself. For example, say you want to take a screenshot, and put the result in a FireMonkey `TBitmap`. As there is no FMX function to do this, one option is to write reams of purely API-level code. Sane people will make use of the VCL `TBitmap` however:

```
uses
  Winapi.Windows, Vcl.Graphics, FMX.Platform;

type
  { Ensure TBitmap without qualification remains the FMX
    TBitmap, and create an alias for the VCL version }
  TBitmap = FMX.Types.TBitmap;
  TVclBitmap = Vcl.Graphics.TBitmap;

procedure TakeScreenshot(Dest: TBitmap);
var
  DC: HDC;
  Size: TPointF;
  VCLBitmap: TVclBitmap;
  Y: Integer;
begin
  VCLBitmap := nil;
  Size := Platform.GetScreenSize;
  DC := GetDC(0);
  try
    VCLBitmap := TVclBitmap.Create;
    VCLBitmap.PixelFormat := pf32bit;
    VCLBitmap.SetSize(Trunc(Size.X), Trunc(Size.Y));
    BitBlt(VCLBitmap.Canvas.Handle, 0, 0, VCLBitmap.Width,
      VCLBitmap.Height, DC, 0, 0, SRCCOPY);
    Dest.SetSize(VCLBitmap.Width, VCLBitmap.Height);
    for Y := Dest.Height - 1 downto 0 do
      Move(VCLBitmap.ScanLine[Y]^, Dest.ScanLine[Y]^, Dest.Width * 4);
  finally
    ReleaseDC(0, DC);
    VCLBitmap.Free;
  end;
end;
```

This works because the in-memory format of a FMX `TBitmap` on Windows is always a 32 bit 'device independent bitmap' (DIB), which is the in-memory format of a VCL `TBitmap` when its `PixelFormat` property is set to `pf32bit`.

Aside from graphics, another area in which the VCL can be useful are dialogs. For example, the FireMonkey `MessageDlg` function uses a custom form, which some might say looks out of place on Windows Vista or later, if not earlier versions too. Add `Vcl.Dialogs` to your uses clause, and the VCL implementation will be available that calls the native equivalent though. Similarly, native folder picker dialogs are available to a FMX/Windows application either via the `SelectDirectory` function of the `Vcl.FileCtrl` unit (this shows the 'classic' Win9x-XP folder picker)...

```
uses Vcl.FileCtrl;

procedure TForm1.Button1Click(Sender: TObject);
var
  Dir: string;
begin
  if SelectDirectory('Select Directory', '', Dir) then
    ShowMessage(Dir);
end;
```

... or the `TFileOpenDialog` class of `Vcl.Dialogs`, which with one property setting shows the Vista+ folder picker:

```
uses Vcl.Dialogs;

procedure TForm1.Button1Click(Sender: TObject);
var
  Dlg: TFileOpenDialog;
begin
  Dlg := TFileOpenDialog.Create(nil);
  try
    Dlg.Options := [fdoPickFolders];
    if Dlg.Execute then
      ShowMessage('You picked ' + Dlg.FileName);
  finally
```

```
    Dlg.Free;
  end;
end;
```

To avoid name clashes when using both `Vcl.Dialogs` and `FMX.Dialogs`, add the VCL unit to the main uses clause immediately *before* the FMX one:

```
uses
  ..., Vcl.Dialogs, FMX.Dialogs, ...
```

This will however mean `MessageDlg` calls go to the FMX version. To always use the VCL version instead, fully qualify with the name of the VCL unit:

```
Vcl.Dialogs.MessageDlg('Native is best', mtInformation, [mbOK], 0);
```

Working with COM APIs

Alongside the classic procedural API on Windows stands the Component Object Model (COM). Dating from the early 1990s, this technology had its day in the sun later on in the decade under the 'ActiveX' branding, before being eclipsed as Microsoft turned its attention elsewhere. Nonetheless, it never went away, remaining a core part of Microsoft Office, APIs such as Direct3D, and latterly, the foundation of the upcoming 'Windows Runtime' (WinRT) of Windows 8.

The basic idea of COM is to provide a subset of OOP functionality in a language neutral fashion. More specifically, it is based upon object interfaces, all of which extend the base COM interface, `IUnknown` — there are no classes, let alone metaclasses, in COM. In reality, talk of language neutrality is a bit misleading, since the COM definition of an interface is not the only possible realisation of the concept. However, in a Delphi context, an equation of COM interfaces with interfaces as such *does* apply, because the Delphi `IInterface` type is exactly the same as the COM `IUnknown`: every Delphi interface, in other words, is a COM interface in a minimal sense.

Because of that, what was said in chapter 4 concerning interfaces in general has relevance to working with COM. In particular, COM objects are 'reference counted' — automatically freed when the last reference is `nil`'ed or goes out of scope — and have their type identity determined by GUIDs.

Beyond the `IUnknown`/`IInterface` identity, Delphi's COM support at the language level has two further aspects: the COM string type ('`BSTR`') is mapped to `WideString`, and the '`safecall`' calling convention exists to convert COM-style error handling to the Delphi exception model. The `BSTR` mapping means you can allocate, use and deallocate `BSTR` instances as if they were they were native Delphi strings. This removes the need to call the COM `BSTR` APIs directly. A further benefit of the `WideString` mapping is that this type supports standard string functions like `Length`, `SetLength` and `SetString`, along with direct assignments to and from native strings, 1-based indexing, and so on.

For example, consider the `QueryPathOfRegTypeLib` API function, which returns the path to a registered type library as a `BSTR`. According to MSDN, 'The caller allocates the `BSTR` that is passed in, and must free it after use'. Since the `WideString` type is compiler managed, this advice doesn't apply in a Delphi context:

```
uses Winapi.ActiveX; //contains the core COM API declarations

const
  Ado20GUID: TGUID = '{00000200-0000-0010-8000-00AA006D2EA4}';
var
  Path: WideString;
begin
  if QueryPathOfRegTypeLib(Ado20GUID, 2, 0, 0, Path) = S_OK then
    WriteLn('The ADO 2.0 type library is at ' + Path)
  else
    WriteLn('Could not find the ADO 2.0 type library');
end.
```

Since a COM `BSTR` uses the same character encoding scheme used by a native Delphi `string`, there is no data loss when one is assigned to another:

```
var
  ComStr: WideString;
  DelphiStr: string;
  UTF8Str: UTF8String;
begin
  ComStr := 'String interop ';
  DelphiStr := ComStr + 'is ';
  ComStr := DelphiStr + 'convenient';
  WriteLn(ComStr); //output: String interop is convenient
```

The safecall calling convention

The `safecall` calling convention concerns how exceptions are not part of COM: instead, the standard is for all routines to return an integral `HRESULT` value, set to 0 (`S_OK` if you prefer a constant) when the call was successful and an error code when something wrong happened. High level COM client languages will then wrap each COM call with a check for whether the `HRESULT` was non-zero, raising an exception (or equivalent) if that is the case.

In Delphi, you can essentially choose which approach to take: declare a COM function using the `stdcall` calling convention and a `HRESULT` return type, and you get the low-level approach; alternatively, use `safecall` and drop the `HRESULT`, and you get the 'high level' way of doing things. Put into code, the following declarations are therefore binary equivalent:

```
type
  ISimpleCOMIntf = interface
  ['{84702B70-C620-0192-290D-F182046A2121}']
```

```
    function SomeProc: HRESULT; stdcall;
    function DoesExist(out RetVal: WordBool): HRESULT; stdcall;
  end;

  ISimpleCOMIntfSC = interface
  ['{84702B70-C620-0192-290D-F182046A2121}']
    procedure SomeProc; safecall;
    function DoesExist: WordBool; safecall;
  end;
```

Generally, use of `safecall` is much more convenient than messing about with `HRESULT` return values. Since it is a language feature, it can also be used independently of COM programming proper, for example when writing custom DLLs — a possibility we will be looking at in the next chapter.

OLE Automation

A popular use of COM is OLE Automation, in which one application exposes special COM interfaces in order to be programmatically controlled by another. The most widely used Automation servers are the constituent programs of Microsoft Office, in particular Word and Excel. For example, Word is frequently used to generate reports or standard letters in database-driven applications.

In the COM jargon, Automation objects are 'dual interface'. This means they can be accessed through either 'early binding' or 'late' binding'. 'Early binding' means interface references are used, as if you were working with Delphi objects through interfaces. In contrast, 'late binding' means a scripting approach is adopted, in which method and property names are looked up dynamically at runtime.

Internally, late binding is based around `IDispatch`, a special interface defined by COM. In Delphi, assigning an `IDispatch` instance to either a `Variant` or `OleVariant` variable allows you to call the object's methods and read or write its properties in much the same way as you might in a scripting language such as VBScript (an `OleVariant` is a `Variant` that will never internally store Delphi-specific data). For example, the following snippet uses late binding to create a new Microsoft Word document containing the text 'Hello world':

```
uses System.Win.ComObj; //for CreateOleObject

procedure CreateWordDoc(const AFileName: string);
var
  WordApp, NewDoc: OleVariant;
begin
  WordApp := CreateOleObject('Word.Application');
  NewDoc := WordApp.Documents.Add;
  NewDoc.Content := 'Hello World';
  NewDoc.SaveAs(FileName);
  WordApp.Quit;
end;
```

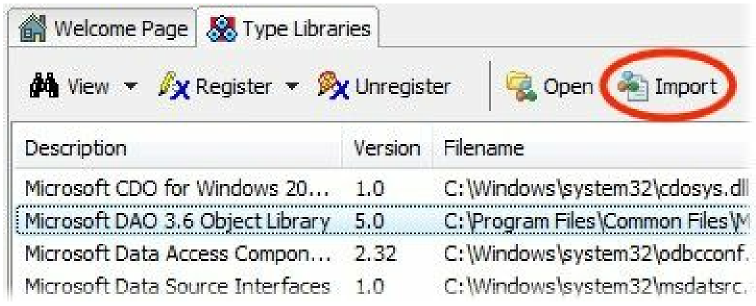
The `CreateOleObject` function is the direct equivalent to `CreateObject` in VBScript or Visual Basic for Applications (VBA). If you prefer, you can also create an Automation object via its GUID using `CreateComObject` — internally, `CreateOleObject` looks up the GUID and calls `CreateComObject`.

Early binding

While late binding can be convenient, it is also error prone and relatively slow — in particular, if you misspell something the compiler will not be able to tell you. Because of this, early binding is usually preferred.

In the case of full-blown COM libraries, early binding is supported by the inclusion of a 'type library'. A COM type library is a binary file (sometimes linked into the DLL, OCX or EXE, but not necessarily) that lists all the types published by the automation server or Active X control. As the Delphi compiler cannot consume type libraries directly, import units need to be generated, similar to how the procedural API requires import units.

Import units for various versions of Microsoft Office, together with a few other things like MSXML and Internet Explorer, are included with Delphi by default — the units for Office are `Word2000`, `WordXP`, `Excel2000`, etc.; for MSXML `Winapi.msxml`; and for Internet Explorer `SHDocVw`. Units for other type libraries can be created in the IDE. To do this, select `View|Registered Type Libraries` from the main menu bar. This will cause a new page to appear in the editor that lists all the registered type libraries on the system. From the toolbar, you can then either browse the contents of a given type library or create an import unit for it:



Description	Version	Filename
Microsoft CDO for Windows 20…	1.0	C:\Windows\system32\cdosys.dll
Microsoft DAO 3.6 Object Library	5.0	C:\Program Files\Common Files\M
Microsoft Data Access Compon…	2.32	C:\Windows\system32\odbcconf.
Microsoft Data Source Interfaces	1.0	C:\Windows\system32\msdatsrc.

As imported, a COM library will be composed of a series of interfaces and constants, together with factory classes for the 'CoClasses' (if you selected the 'component wrappers' option in the import wizard, there will also be some simple `TComponent` descendants to install into the Tool Palette too — personally I find them a little pointless in practice). Ideally the type library will be designed so that the interface and CoClass names match up. Thus, if a COM library called `FooLib.dll` exports an interface called `IFoo`, you would use the `CoFoo` factory class to create an instance of it:

```
uses FooLib_tlb.pas;

var
  Foo: IFoo;
begin
  Foo := CoFoo.Create;
  //... use Foo
end;
```

However, not only is it up to the COM server to determine which of its interfaces correspond to externally creatable objects (perhaps FooLib defines another interface, `IBar`, that isn't creatable), but CoClass and interface name are in principle independent of each other. Similarly, while the name of a COM interface, like a Delphi one, normally begins with a capital 'I', it need not — the names of many Microsoft Office interfaces begin with a leading underscore instead, for example.

Office hassles

On the face of it, you might think the object models of Microsoft Office would be archetypes of how OLE Automation servers should be written. Since they evolved while OLE Automation itself was evolving however, with different Office programs acquiring full-blown OLE Automation support at different times, this is not really the case. In practical terms, this means there are hassles involved in using Delphi or even Microsoft's own .NET languages to program Office that you don't have when using VBA, i.e. programming Office from within Office itself.

In particular, early binding frequently necessitates passing `EmptyParam` repeatedly when calling a method:

```
uses
  Word2000; //earliest to maximise backwards compatibility

procedure CreateWordDoc(const AFileName: string);
var
  WordApp: WordApplication;
  NewDoc: WordDocument;
begin
  WordApp := CoWordApplication.Create;
  NewDoc := WordApp.Documents.Add(EmptyParam, EmptyParam,
   EmptyParam, EmptyParam);
  NewDoc.Content.Text := 'Hello World';
  NewDoc.SaveAs(FileName, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam);
  WordApp.Quit(False, False, False);
end;
```

`EmptyParam` is a function from `System.Variants` that returns a `Variant` containing the special value OLE Automation defines for saying, in effect, 'please use whatever you consider to be the default value'.

Another Office annoyance is how various methods have string inputs that nevertheless use parameters of the form

```
var SomeName: OleVariant
```

In such a case, just do what the parameter type implies: define a local variable typed to `OleVariant`, assign the string to it, and pass the variant to the method.

Going further with Delphi's COM support

What I’ve described in the previous few pages really only scratches the surface of Delphi’s support for COM: using tooling providing by the IDE, you can create your own Automation servers, COM+ components, ActiveX controls, and more. Looking at these would take us beyond the scope of this book however. Instead, we will turn to XE2’s second major target platform: OS X.

Mac OS X

The operating system APIs on OS X are split into several levels, most notably the 'POSIX', 'Core' and 'Cocoa' layers. The first two have a straight C interface, the third an Objective-C one. We'll look at each in turn.

POSIX

The POSIX API relates to OS X's Unix core, and forms a standardised, if somewhat low-level interface that you can find replicated in other Unix or Unix-like operating systems (e.g. Linux). As such, the API style is not entirely dissimilar to the traditional Windows API. However, one difference is that POSIX uses UTF-8 for string data where Windows uses UTF-16 or 'Ansi'. For example, the POSIX function to retrieve the name of the current user is `getlogin`. For a C/C++ audience, this function is declared in `unistd.h`, and looks like this:

```
char *getlogin(void);
```

As translated into Delphi, `getlogin` becomes the following:

```
function getlogin: PAnsiChar; cdecl;
```

Similar to import units for the Windows API, ones for POSIX are named after the original C header files and given a `Posix.*` prefix (there is no general dumping ground for commonly-used POSIX functions, like the `Windows.Winapi` unit). Thus, calling `getlogin` in Delphi requires adding `Posix.Unistd` to the uses clause. Since the function returns a pointer to a buffer maintained on the system side, we don't need to explicitly allocate anything in Delphi — instead, just call the `UTF8ToUnicodeString` RTL function to convert the returned data to UTF-16:

```
uses Posix.Unistd;

begin
  WriteLn('Your user name is ' + UTF8ToUnicodeString(getlogin));
end.
```

In the case of a POSIX routine that requires you to send rather than receive a string, use a double cast, first to `UTF8String` and then to `PAnsiChar`:

```
uses Posix.Stdlib;

procedure SetEnvironmentVariable(const AName, AValue: string);
begin
  setenv(PAnsiChar(UTF8String(AName)),
    PAnsiChar(UTF8String(AValue)), 1);
end;
```

Using the 'Core' API layer

Like the POSIX layer, the 'Core' APIs are designed to be callable by a program written in C. As such, they pose no special interoperability problems for Delphi. Nonetheless, in terms of feel they are quite different to the POSIX API. This is because, while formally procedural, they work in terms of reference counted objects. In Apple's jargon, many are also 'toll free bridged' with the equivalent Cocoa types, meaning most 'Core' objects actually *are* Cocoa objects under the hood.

In XE2, translations for OS X specific APIs are provided under the `Macapi` unit scope. This means declarations for the most fundamental 'Core' API, Core Foundation, are found in the `Macapi.CoreFoundation` unit. In the following example, it is put to use to show a native message box:

```
uses Macapi.CoreFoundation;

procedure ShowMessageCF(const AHeading, AMessage: string;
  const ATimeoutInSecs: Double = 0);
var
  LHeading, LMessage: CFStringRef;
  LResponse: CFOptionFlags;
begin
  LHeading := CFStringCreateWithCharactersNoCopy(nil,
    PChar(AHeading), Length(AHeading), kCFAllocatorNull);
  LMessage := CFStringCreateWithCharactersNoCopy(nil,
    PChar(AMessage), Length(AMessage), kCFAllocatorNull);
  try
    CFUserNotificationDisplayAlert(ATimeoutInSecs,
      kCFUserNotificationNoteAlertLevel, nil, nil, nil,
      LHeading, LMessage, nil, nil, nil, LResponse);
  finally
    CFRelease(LHeading);
    CFRelease(LMessage);
  end;
end;

begin
  ShowMessageCF('Test', 'I will auto-destruct in 10 secs!', 10);
end.
```

This example illustrates a couple of things in particular. The first is the object-oriented nature of the API: rather than just pass in `PChar` values, you must construct `CFStringRef` objects. Secondly, it demonstrates the manual reference counting model. The basic principle of this is the following: if you receive a Core Foundation object via a function with `Create` or `Copy` in its name, you must call `CFRelease` once you have finished with it (Apple refers to this as the 'create rule'). Otherwise, the object returned is considered 'owned' by the returner (the 'get rule').

If you receive a CF object through the 'get rule' and want to keep hold of it, you must call `CFRetain`, acquiring 'ownership' yourself. Internally, calling `CFRetain` increments the object's 'retain' or reference count, and must be paired with an explicit call to `CFRelease` once the object is no longer needed:

```
var
  S: CFStringRef;
begin
  //...
  CFRetain(S);
  try
    //...
  finally
    CFRelease(S);
  end;
```

Conceptually, `CFRetain` and `CFRelease` are similar to the `_AddRef` and `_Release` methods supported by every COM or Delphi interface (we met `_AddRef` and `_Release` when looking at `IInterface` in chapter 4). The main difference is that `CFRetain` and `CFRelease` are not called automatically, in contrast to `_AddRef` and `_Release`.

Interfacing with Objective-C and the Cocoa API

As the Core Foundation layer has quite a different feel to the POSIX one, so the Cocoa API is quite different again. This is because it is designed for (and largely implemented in) Objective-C, an object-oriented superset of C. The practical effect of this is that while POSIX and the 'Core' APIs can be called directly from Delphi, some sort of special mechanism is necessary for Delphi code to call into Cocoa.

Officially, Delphi XE2 targets both OS X and iOS, Apple's desktop and mobile/tablet operating systems respectively. However, iOS is supported via the Free Pascal Compiler (FPC), an open source alternative to Delphi, rather than the actual Delphi compiler and RTL. Unfortunately, the way the two products enable calling Objective-C APIs is fundamentally different. Since this book is about Delphi, I will only be looking at the approach taken for OS X. If you need to understand the FPC/iOS one, an excellent resource beyond the official FPC documentation is 'Developing with Objective Pascal', a free set of articles by Phil Hess (`http://web.me.com/macpgmr/ObjP/Xcode4/ObjP_Intro.html`).

Objective-C: a 30 second primer

Objective-C primarily works in term of classes, metaclasses and 'protocols', the last of which roughly correspond to interfaces in Delphi. Calling a method is referred to as sending a message, with message resolution always being performed at runtime. This means 'static' method calls, which Delphi uses by default unless you explicitly mark a method either `virtual` or `dynamic`, do not exist.

In principle, you can code in Objective-C without touching the Cocoa API — indeed, the language predates it by some years. However, even pure Objective-C programming normally works with objects descending from `NSObject`, the base class in the Cocoa framework ('NS' stands for NeXTSTEP, the non-Apple operating system that became OS X). This is because Objective-C has no intrinsic common root class itself, leaving `NSObject` to play a similar role to `TObject` in Delphi.

Slightly confusingly at first, there is also an `NSObject` protocol that the `NSObject` class implements. In Objective-C, this is just to allow calling the `NSObject` methods on a protocol reference, since by default, a bare protocol type won't define them, just like a bare interface type in Delphi doesn't define the `TObject` methods (i.e., `ClassName`, `ToString`, etc.).

As in Delphi, creating a new object in Objective-C involves a two-stage process: first the memory for it is dynamically allocated, before its fields and so forth are initialised. However, while Delphi's constructor syntax tends to hide the fact of there being two stages, this is much less the case in Objective-C.

To allocate the memory for a new object, you call `alloc`, and to initialise it, `init` or variant thereof; in Delphi terms, `alloc` is equivalent to `NewInstance`, and `init` is equivalent to calling a constructor as an instance method (normally you call a Delphi constructor as a class method, which has `NewInstance` called implicitly).

In Objective-C itself, if an object is to be constructed via the inherited `alloc` (which is the norm) followed by a call to the parameterless `init`, then the `new` keyword can be used to syntactically merge the two calls into one. However, since the `new` keyword only works with the parameterless `init`, use of any other initialiser will require an explicit `alloc`/`initXXX` pair. This contrasts with the constructor syntax in Delphi, which in Objective-C terms works with custom 'initialisers' of any name and with any parameter list.

Technically, the core OOP functionality of Objective-C is implemented by a runtime library that is callable by any language that can work with a simple C compatible API. In Delphi, import declarations for it are declared in the `Macapi.ObjCRuntime` unit. The number of routines exported by the runtime is surprisingly small — there's only a couple dozen functions in all — and gives a hint to how the distinctive feel of Cocoa programming derives less from the Objective-C language as such, than the conventions Cocoa and related frameworks overlay it with.

Modelling Objective-C class types in Delphi

XE2's Delphi to Objective-C bridge works by exposing Objective-C classes, metaclasses and protocols as interface types, named after the Objective-C originals. Thus, corresponding to the `NSString` class in Objective-C are two Delphi interface types: `NSString`, which contains the Objective-C instance methods, and `NSStringClass`, which contains the class methods. Technically, the interface types are wrappers — when you call one of their methods, you aren't calling the Objective-C method directly, but asking the bridging code to send a message that corresponds to it.

Aside from the interface types, each exposed Objective-C class also has a corresponding Delphi class, named with a `T` prefix, that acts as a simple factory. The factory class contains two core members: a class function called `Create`, and another called `OCClass`. Where the first returns a new, 'wrapped' instance of the Objective-C class, the second returns the wrapped Objective-C metaclass. In effect, calling `Create` provides an analogue to the `new` keyword in Objective-C itself, calling `alloc` and the parameterless `init`:

```
uses Macapi.Foundation;

var
  Task: NSTask;
begin
  Task := TNSTask.Create;
  try
    Task.setLaunchPath(NSSTR('/Applications/TextEdit'));
    //etc.
    Task.launch;
    Task.waitUntilExit;
  finally
    Task.release;
  end;
```

In practice, you don't actually call `TSomeFactory.Create` as much as you might expect. This is due to how the authors of the Cocoa frameworks had a predilection for class methods that return either new or shared ('singleton') instances. For example, to retrieve an `NSApplication` instance, you call the `sharedApplication` class method; to return an `NSFileManager`, you call the `defaultManager` class method; and instead of multiple constructor overloads, as you would expect in an equivalent Delphi class, the metaclass for `NSString` has helper methods such as `stringWithUTF8String`.

Unfortunately, a limitation of the Delphi to Objective-C bridge as it currently stands is that these class methods return 'raw', unwrapped Objective-C object references. You can tell this is the case since the return types are `Pointer` rather than an interface of some sort (`NSString` or whatever). Nonetheless, any raw object reference can be wrapped via the `Wrap` class function of the corresponding Delphi factory class.

For example, `NSTask` has a convenience class method called `launchedTaskWithLaunchPath`. Calling it from Delphi goes like this:

```
var
  Task: NSTask;
begin
  Task := TNSTask.Wrap(
    TNSTask.OCClass.launchedTaskWithLaunchPath(Path, Args));
```

This technique of calling `Wrap` works just as well with references to Core Foundation objects that are 'toll free bridged' with Cocoa equivalents:

```
uses
  Macapi.CoreFoundation, Macapi.Foundation;
```

```
procedure UseSomeCocoaFunc(CFStr: CFStringRef);
var
  NSStr: NSString;
  Value: Integer;
begin
  NSStr := TNSString.Wrap(CFStr);
  Value := NSStr.intValue;
  WriteLn('Converting to an integer using Cocoa gives ', Value);
end;
```

To go the other way — i.e., to get a raw Objective-C object from a Delphi interface wrapper — use the `Macapi.ObjectiveC` unit, query for the `ILocalObject` interface, and call its `GetObjectID` method:

```
uses Macapi.ObjectiveC;

var
  RawObj: Pointer;
begin
  RawObj := (SomeNSIntf as ILocalObject).GetObjectID;
```

This technique can be handy for writing helper functions that convert to and from native Delphi types, as it means a Cocoa version can be a one liner that delegates to the Core Foundation variant:

```
uses
  Macapi.CoreFoundation, Macapi.ObjectiveC, Macapi.CocoaTypes,
  Macapi.Foundation;

function CFToDelphiString(const CFStr: CFStringRef): string;
var
  Range: CFRange;
begin
  if CFStr = nil then Exit('');
  Range.location := 0;
  Range.length := CFStringGetLength(CFStr);
  SetLength(Result, Range.length);
  if Range.length = 0 then Exit;
  CFStringGetCharacters(CFStr, Range, PWideChar(Result));
end;

function NSToDelphiString(const S: NSString): string; inline;
begin
  if S <> nil then
    Result := CFToDelphiString((S as ILocalObject).GetObjectID)
  else
    Result := '';
end;
```

Cocoa reference counting

Like Core Foundation, Cocoa has a reference counting model — as Core Foundation has `CFRetain`, `CFRelease` and `CFRetainCount` routines, so `NSObject` has `retain`, `release` and `retaincount` methods. However, Cocoa is complicated (or simplified, depending on your point of view!) by the concept of 'auto-release pools': when an object is added to such a pool, it will have its `release` method called when the pool itself is released. Internally, auto-release pools are allocated into a system-managed stack (i.e., first in, first out list). This means you don't usually specify which pool an object is put into: rather, it goes into the pool that was most recently created.

Auto-release pools play a big part in Cocoa due to a convention of putting objects created using a class method into the top auto-release pool. Consequently, explicit calls to `release` on a Cocoa object are much rarer than explicit calls to `CFRelease` on a Core Foundation one. To take a trivial example, the following `NSString` will therefore usually not need its `release` method explicitly called:

```
uses Macapi.Foundation;

var
  S: NSString;
begin
  S := TNSString.Wrap(
    TNSString.OCClass.stringWithUTF8String('This is a test'));
```

Internally, Cocoa-based GUI applications (FireMonkey programs included) create and destroy a new auto-release pool at the start and end of each event loop iteration. So, when the user clicks the mouse (for example), the system first creates an auto-release pool, secondly gives the application a chance to handle the click (in a FireMonkey application, via the relevant `OnXXX` event property), before finally releasing the pool, causing any objects added to it to be released too.

Outside the main thread of a GUI application, auto-release pools are however *not* automatically created. This means you must create one yourself when writing either a console application or a secondary thread that may use auto-released Cocoa objects:

```
uses Macapi.Foundation;

var
  AutoReleasePool: NSAutoreleasePool;
begin
  //reference *something* to ensure Cocoa framework is loaded
  NSDefaultRunLoopMode;
  //once loaded, can instantiate a Cocoa class...
  AutoReleasePool := TNSAutoreleasePool.Create;
  try
    //create and use auto-released Cocoa objects here...
  finally
    AutoReleasePool.drain;
  end;
end.
```

The use of `drain` (which is particular to `NSAutoreleasePool`) rather than `release` here is technically arbitrary, though Apple recommends it.

Declaring custom Cocoa imports

While XE2 ships with some import declarations for Cocoa — in particular, `Macapi.Foundation` contains translations for things like `NSString`, and `Macapi.AppKit` for things like `NSApplication` and `NSWindow` — the coverage is not very comprehensive compared to what is provided for both the Windows and POSIX APIs. On occasion, you therefore may need to write custom imports. Doing so has three main steps:

- Declare interface types for the Objective-C class and metaclass. These should be parented appropriately, e.g. to `NSObject` and `NSObjectClass` respectively. The types should also be given GUIDs, however what GUIDs exactly doesn't matter — just give them unique ones. (In the IDE editor, allocate a new GUID by pressing `Ctrl+Shift+G`.)
- Add the required methods to the interface types. These can be listed in any order, and you only need declare the methods you actually need to call. However, method and parameter names must match the Objective-C originals exactly, and all methods must be declared with the `cdecl` calling convention.
- Declare the factory class by deriving a simple descendant of `TOCGenericImport` that instantiates the generic with the two interface types you just defined.

To give an example of this, at my time of writing, the `QTMovie` class does not have a standard import. This is a shame, since despite its name, `QTMovie` serves as a very easy way to play sound files. To use it for that purpose, you need only surface three methods in total: `movieWithFile` (which is a class method), `play` and `stop`. Here's the entire import code for that:

```
uses
  Macapi.ObjectiveC, Macapi.CocoaTypes, Macapi.Foundation;

type
  QTMovieClass = interface(NSObjectClass)
    ['{92E2B523-AA56-4D7F-81B8-CFB56D548DDD}']
    function movieWithFile(fileName: NSString;
      error: PPointer = nil): Pointer; cdecl;
  end;

  QTMovie = interface(NSObject)
    ['{754E82A2-0135-4805-A7FE-D3A7B49ACC37}']
    procedure play; cdecl;
    procedure stop; cdecl;
  end;

  TQTMovie = class(TOCGenericImport<QTMovieClass, QTMovie>);
```

In use, construct a new instance via the `movieWithFile` class method, retain it (this is a QTMovie quirk) and call `play`. While playing, call `stop` if you want to be able to call `play` again or `release` if you have finished with the object:

```
procedure TfrmQTMovie.btnOpenAndPlayFileClick(Sender: TObject);
begin
  if not OpenDialog1.Execute then Exit;
  if FMovie <> nil then
  begin
    FMovie.release;
    FMovie := nil;
  end;
  FMovie := TQTMovie.Wrap(TQTMovie.OCClass.movieWithFile(
    NSSTR(OpenDialog1.FileName)));
  FMovie.retain;
  FMovie.play;
end;

procedure TfrmQTMovie.FormDestroy(Sender: TObject);
begin
  if FMovie <> nil then FMovie.release;
end;
```

Translating Objective-C method declarations

The hard part of custom Cocoa imports is not actually using them, but getting the method declarations correct beforehand. Apple's documentation is strongly oriented towards Objective-C, but in many cases converting to Delphi Pascal is not too hard:

- If in the documentation the Objective-C declaration is prepended with a plus sign (+), it is a class method and so should be added to the Delphi interface that represents the metaclass (in the example just given, `QTMMovieClass`). If it is prepended with a minus sign (-), in contrast, it is an instance method and should be added to the Delphi interface that represents the class (`QTMovie`).
- The return type of an Objective-C method is declared immediately after the plus or minus; when the return type is `void`, the method should be declared a procedure in Delphi. Thus, the original Objective-C declaration of `play` looks like this:

```
 - (void)play
```

- For each parameter, the type comes before the name, and is enclosed in brackets (this contrasts to Pascal, in which the name comes first).
- If the method declaration uses simple types, check the needed type aliases haven't already been declared in the `Macapi.CocoaTypes` unit. This defines `NSInteger`, amongst others.
- A parameter or return type prepended by an asterisk denotes a pointer to an instance of the specified type (prepended by *two* asterisks means a pointer to a pointer). In Delphi, this can become either a pointer type similarly (the most direct translation would be to prepend the type name with a ^), or in the case of a parameter, a `var` parameter. So, if a parameter is declared in Objective-C as

```
 (NSInteger *) someInt
```

  it becomes in Delphi either

```
 someInt: ^NSInteger
```

  or

```
 someInt: PNSInteger
```

  or

```
 var someInt: Integer
```

- An exception to the previous rule is when the type is a pointer to an object: if a Delphi wrapper interface is defined for the Objective-C class involved, *don't* type the parameter to a pointer to the interface — just make it typed to the interface. The reason for this discrepancy is that objects in Objective-C require explicit referencing and dereferencing, unlike in Delphi (or indeed, the historical Apple dialect of Object Pascal!).
- Again with object parameters, you can if you prefer type them to `Pointer` (representing an unwrapped Objective-C object) rather than an interface of some sort (representing a wrapped Objective-C object). Alternatively, if the Objective-C class is 'toll-free bridged' with a Core Foundation type (e.g., `NSString` is toll-free bridged with `CFStringRef`, `NSArray` with `CFArrayRef` and so on), you can type the parameter to the Core Foundation equivalent. Consequently, the following Objective-C method declaration

```
 - (void)foo:(NSString *) someName
```

  could be translated in three different ways:

```
 procedure foo(someName: NSString); cdecl;
 procedure foo(someName: Pointer); cdecl;
 procedure foo(someName: CFStringRef); cdecl;
```

- In Delphi, method overloads are distinguished by parameter *types*; in Objective-C, in contrast, they are distinguished by parameter *names*. This can cause a set of overloads legal in Objective-C to be illegal in Delphi (or vice versa). When this happens, the workaround is to declare one of the overloads using the wrapper `NSxxx` type and the other using the unwrapped `CFxxxRef` type, or `Pointer`, or a new type based on `Pointer` (or `CFxxxRef`):

```
type
  Pointer2 = type Pointer;
  CFStringRef2 = type CFStringRef;

  NSSomething = interface
  [GUID]
    procedure sayHi(sender: Pointer;
      rawName: NSString); cdecl; overload;
    procedure sayHi(sender: Pointer;
      title: CFStringRef); cdecl; overload;
    procedure sayHi(sender: Pointer;
      bundleName: CFStringRef2); cdecl; overload;
    procedure sayHi(sender: Pointer;
      identifier: Pointer); cdecl; overload;
    procedure sayHi(id: Pointer;
      mangledName: Pointer2); cdecl; overload;
  end;
```

Implementing Cocoa callbacks

On occasion, an API routine may take as a parameter a 'callback', or function of your own that is then called by the API. Passing a callback in the case of the classic Windows API or even Core Foundation is pretty easy. This is because both will require just a pointer to a standalone routine, or at most a group of such pointers. Writing Cocoa callbacks (called 'delegates' in Apple's terminology) is a little more complicated however, since you need to implement an Objective-C class containing the required method or methods.

In XE2, this is done by descending from the `TOCLocal` class (this is defined in the `Macapi.ObjectiveC` unit), and implementing a Delphi interface that represents the Objective-C delegate class. This interface should be descended from `NSObject`, and be composed of methods that all have the `cdecl` calling convention. When implementing it, it is advisable not to *formally* do so — i.e., do not list the interface as one the class implements in its header. Instead, override the protected `GetObjectiveCClass` method in the following manner:

```
uses
  System.TypInfo, Macapi.ObjectiveC, Macapi.CocoaTypes;

type
  MyDelegate = interface(NSObject)
    ['{78E2B523-AA56-4D7F-81B8-CFB56D548AAA}']
    procedure someMethod; cdecl;
  end;

  TMyDelegate = class(TOCLocal) //do NOT list MyDelegate here!
  protected
    function GetObjectiveCClass: PTypeInfo;
    procedure someMethod; cdecl;
  end;

function TMyDelegate.GetObjectiveCClass: PTypeInfo;
begin
  Result := TypeInfo(MyCocoaDelegate);
end;
```

The idea here is that `GetObjectiveCClass` determines the Objective-C class represented by the Delphi type, whereas any formally implemented interfaces represent the supported Objective-C protocols.

Having written the delegate class (we will look at a complete example shortly), the client object may require a 'selector' as well as a pointer to the delegate object itself. Roughly speaking, a 'selector' is a pointer to something that identifies the method to call, and can be retrieved by calling the `sel_getUid` function from the Objective-C runtime. This function is passed a string with the form

```
methodName:secondParamName:thirdParamName:andSoOn:
```

Important here is how the method's first parameter isn't listed, and the string always ends in a colon. Be careful to get the casing exactly right too — it should match the casing you used when declaring the interface type for the delegate class.

Another thing to watch out for are memory management issues. These stem from how the delegate's implementation will come in two parts, one done on the Delphi side and the other (the implementation of `NSObject`) in Objective-C. Thus, the Objective-C part has its own reference count (incremented and decremented by the `retain` and `release` messages, or

`CFRetain` and `CFRelease` routines) that *doesn't* control the lifetime of the Delphi portion; likewise, calling `Destroy` or `Free` on the Delphi object does not in itself free the Objective-C core.

If you need to write Objective-C classes for general use, Delphi is not, therefore, a good tool to use — funnily enough, Objective-C itself would be a much better bet. However, it is good enough for writing delegates that are ultimately under the control of surrounding Delphi code. Let's look at a practical example of this: writing a delegate for the `NSAlert` class.

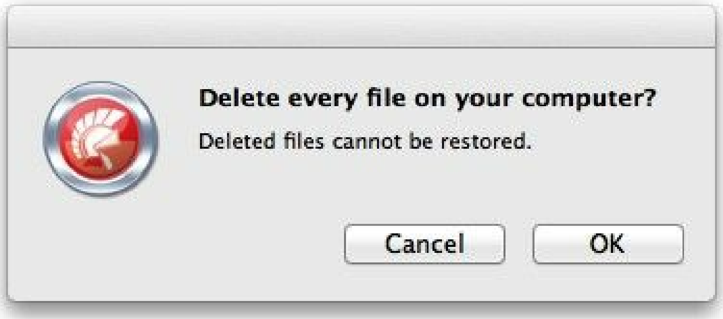
Example: writing a delegate for the NSAlert class

`NSAlert` is the Cocoa way to show a simple message box. It supports both the traditional dialog style, in which the message is displayed apart from the host application in the middle of the screen, and Apple's preferred way of a 'modal sheet', in which the message is shown in a box that slides down from the active form's title bar. In both cases the message is shown 'modally', however in the modal sheet case the modality is 'only' with respect to the host window, not the application as a whole.

That said, when the traditional style is used with `NSAlert`, no delegate is required:

```
uses Macapi.CocoaTypes, Macapi.Foundation, Macapi.AppKit;

procedure TForm1.Button1Click(Sender: TObject);
var
  Alert: NSAlert;
begin
  Alert := TNSAlert.Create;
  try
    Alert.addButtonWithTitle(NSSTR('OK'));
    Alert.addButtonWithTitle(NSSTR('Cancel'));
    Alert.setMessageText(NSSTR('Delete every file on disk?'));
    Alert.setInformativeText(NSSTR('Deleted files cannot ' +
      'be restored.'));
    Alert.setAlertStyle(NSWarningAlertStyle);
    case Alert.runModal of
      NSAlertFirstButtonReturn: Caption := 'You pressed OK';
      NSAlertSecondButtonReturn: Caption := 'You pressed Cancel';
    else
      Caption := 'Er, something went wrong here...';
    end;
  finally
    Alert.release;
  end;
end;
```

This code results in a dialog that looks like this:


Delete every file on your computer?

Deleted files cannot be restored.

Using the modal sheet style instead means replacing the call to `runModal` with one to `beginSheetModalForWindow`. This takes references to the parent window and the delegate object, together with a selector and an pointer for anything you like (this last pointer will then be passed, unprocessed, to the delegate method):

```
procedure beginSheetModalForWindow(window: NSWindow;
  modalDelegate: Pointer; didEndSelector: SEL;
  contextInfo: Pointer); cdecl;
```

To keep things tidy, we will write a helper routine called `ShowNSAlertAsSheet` that takes an anonymous method to use for the delegate; this helper will then invoke `beginSheetModalForWindow` for the caller. Our aim will be to make the call to `runModal` in the example above more or less directly replaceable with one to our helper routine:

```
ShowNSAlertAsSheet(Alert, Self,
  procedure (ReturnCode: NSInteger)
  begin
    case ReturnCode of
      NSAlertFirstButtonReturn: Caption := 'You pressed OK';
      NSAlertSecondButtonReturn: Caption := 'You cancelled';
    else
      Caption := 'Er, something went wrong here...';
    end;
    Alert.release;
  end);
```

Implementing the delegate

So, create a new unit, and change its `interface` section to look like this:

```
interface

uses
  System.TypInfo, Macapi.CoreFoundation, Macapi.CocoaTypes,
  Macapi.AppKit, FMX.Forms;

type
  TNSAlertClosedEvent = reference to procedure (ReturnCode: NSInteger);

procedure ShowNSAlertAsSheet(const Alert: NSAlert;
  Form: TCommonCustomForm; const OnClose: TNSAlertClosedEvent);
```

Since a modal sheet is attached to a window, the caller will need to specify which form to attach to. However, `NSAlert` expects a Cocoa `NSWindow`, not a FireMonkey `TForm`. Since under the bonnet a FireMonkey form on OS X *is* an `NSWindow`, converting between the two should be easy — the VCL analogue would be to read off the form's `Handle` property to get an `HWND`. However, as we have already seen, the FireMonkey framework makes it annoyingly hard to access underlying native API handles, though luckily the `TForm`/`NSWindow` case is one of the easier ones. The following code may look a bit odd (dare I say hacky?), but it works:

```
uses
  Macapi.ObjCRuntime, Macapi.ObjectiveC, FMX.Platform.Mac;

type
  TOCLocalAccess = class(TOCLocal);

function NSWindowOfForm(Form: TCommonCustomForm): NSWindow;
var
  Obj: TOCLocal;
begin
  Obj := (FmxHandleToObjC(Form.Handle) as TOCLocal);
  Result := NSWindow(TOCLocalAccess(Obj).Super);
end;
```

Add this code to the `implementation` section of the unit just created, and we can turn to the delegate itself.

According to Apple's documentation, the delegate must have a method with the following signature:

```
- (void) alertDidEnd:(NSAlert *)alert returnCode:(NSInteger)returnCode contextInfo:(void *)contextInfo;
```

Translated into Delphi Pascal, this becomes as thus:

```
procedure alertDidEnd(alert: NSAlert; returnCode: NSInteger;
  contextInfo: Pointer); cdecl;
```

Given that, what we need to do is to define an interface type deriving from `NSObject` that contains this method — this will represent our delegate Objective-C class — and a Delphi class descended from `TOCLocal` to implement it. In the unit, declare them like this, immediately below the `NSWindowOfForm` function just added:

```
type
  IAlertDelegate = interface(NSObject)
    ['{E4B7CA83-5351-4BE7-89BB-25A9C70D89D0}']
    procedure alertDidEnd(alert: Pointer; returnCode: NSInteger;
      contextInfo: Pointer); cdecl;
  end;

  TAlertDelegate = class(TOCLocal)
  strict private
    FOnClose: TNSAlertClosedEvent;
  protected
    function GetObjectiveCClass: PTypeInfo; override;
  public
    constructor Create(const AOnClose: TNSAlertClosedEvent);
    destructor Destroy; override;
    procedure alertDidEnd(alert: Pointer; returnCode: NSInteger;
      contextInfo: Pointer); cdecl;
  end;
```

Given the sole purpose of the delegate object is to invoke the method reference passed to it when appropriate (i.e., in `alertDidEnd`), the implementation of `TAlertDelegate` is pretty trivial:

```
constructor TAlertDelegate.Create(
  const AOnClose: TNSAlertClosedEvent);
begin
  inherited Create;
  FOnClose := AOnClose;
end;

destructor TAlertDelegate.Destroy;
begin
  CFRelease(GetObjectID);
  inherited;
end;

function TAlertDelegate.GetObjectiveCClass: PTypeInfo;
begin
  Result := TypeInfo(IAlertDelegate);
end;

procedure TAlertDelegate.alertDidEnd(alert: Pointer;
  returnCode: NSInteger; contextInfo: Pointer);
begin
  FOnClose(returnCode);
  Destroy;
end;
```

To ensure the Objective-C part of the class isn't left hanging around, we explicitly release it when the Delphi part is destroyed — calling `CFRelease` does the same thing as sending the `release` method, which we don't have direct access to. At the end of `alertDidEnd`, the Delphi object then frees itself. This may look a bit odd, but there is nowhere else appropriate: we cannot free the delegate object once `beginSheetModalForWindow` returns, since it returns immediately, i.e. while the sheet is still being shown. Furthermore, we can't rely on normal `IInterface` reference counting either, since that depends on the interface reference remaining in scope while it is still needed. A process of elimination therefore leaves us with the delegate freeing itself when it knows it has done the job it was created for.

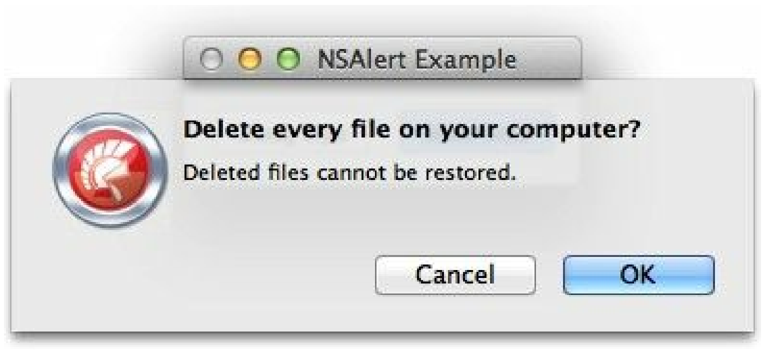
Here's the implementation of `ShowNSAlertAsSheet`. As previously discussed, it uses the `sel_getUid` function to return the 'selector':

```
procedure ShowNSAlertAsSheet(const Alert: NSAlert;
  Form: TCommonCustomForm; const OnClose: TNSAlertClosedEvent);
var
  Delegate: TAlertDelegate;
begin
  Delegate := TAlertDelegate.Create(OnClose);
  Alert.beginSheetModalForWindow(NSWindowOfForm(Form),
    Delegate.GetObjectID,
    sel_getUid('alertDidEnd:returnCode:contextInfo:'), nil);
end;
```

Finally, here's the completed unit put to use, adapting the earlier example for `runModal`:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Alert: NSAlert;
begin
  Alert := TNSAlert.Create;
  Alert.addButtonWithTitle(NSSTR('OK'));
  Alert.addButtonWithTitle(NSSTR('Cancel'));
  Alert.setMessageText(NSSTR('Delete every file on disk?'));
  Alert.setInformativeText(NSSTR('Deleted files cannot ' +
    'be restored.'));
  Alert.setAlertStyle(NSWarningAlertStyle);
  ShowNSAlertAsSheet(Alert, Self,
    procedure (ReturnCode: NSInteger)
    begin
      case ReturnCode of
        NSAlertFirstButtonReturn: Caption := 'You pressed OK';
        NSAlertSecondButtonReturn: Caption := 'You cancelled';
      else
        Caption := 'Er, something went wrong here...';
      end;
      Alert.release;
    end);
end;
```

This then produces a modal sheet that looks like this:



Delete every file on your computer?

Deleted files cannot be restored.

14. Writing and using dynamic libraries

Dynamic libraries — DLLs on Windows, dylibs on OS X — form a way to share compiled code between programs written in different languages or compilers. Any application you write will use them in some form, since the native APIs of the operating system will be implemented as a series of dynamic libraries that the application calls into, if not explicitly in your own code then implicitly via the Delphi RTL.

When it comes to writing custom libraries, an alternative to plain DLLs or dylibs in Delphi are packages, a special sort of Delphi-specific dynamic library we met in chapter 10. Compared to ordinary libraries, packages can be much more convenient to program with. In particular, where plain libraries can export just a series of standalone routines, packages allow maintaining a class-based approach. However, as soon as the need arises to allow application and library to be compiled with different versions of Delphi, let alone allow the application not to be written in Delphi at all, plain libraries come into their own.

Writing a dynamic library

In practical terms, creating a dynamic library means creating a DLL project in the IDE. This generates a DPR file, much like the DPR for a normal project, only with a different keyword at the top (`library` for `program`) and an `exports` section added near the bottom (it will also include a large comment near the top that provides some advice last current in the 1990s. Ignore it.). The `exports` clause lists the names of procedures or functions that will be callable by outside code:

```
library MyLibrary;

function AddThem(Num1, Num2: Int32): Int32;
begin
  Result := Num1 + Num2;
end;

function MultiplyThem(Num1, Num2: Int32): Int32;
begin
  Result := Num1 * Num2;
end;

exports
  AddThem,
  MultiplyThem;

begin
  //any initialisation code goes here...
end.
```

While anything listed under `exports` must be a standalone routine not attached to any class or record type, it need not be implemented in the DPR itself.

When targeting Windows (32 or 64 bit), a library project called `MyLibrary.dpr` will compile to `MyLibrary.dll`; when targeting the Mac, `libMyLibrary.dylib` will be outputted. As a `dll` extension is the Windows norm, so a `lib` prefix and a `dylib` extension are the OS X convention; there's usually little need to do differently.

By default, routines are exported using their real names. However, you can specify something else by adding a `name` part to the `exports` clause:

```
exports
  AddThem name 'AddNums',
  MultiplyThem name 'MultiplyNums';
```

As we shall see, this facility has particular utility when targeting the Mac. However, it can also be useful if you wish to export overloaded routines, since as exported, functions or procedures must be uniquely named.

A further option is to export not by name, but by number. This is done via the `index` keyword:

```
exports
  AddThem index 42;
```

This facility is rarely used nowadays however.

Things to think about when writing custom libraries

If you want your libraries to be callable from as broad a range of possible clients as possible, you need to keep your discipline over three things in particular: adopting an appropriate 'calling convention', exception safety, and using language-neutral types. We'll look at each area in turn.

Calling conventions

A 'calling convention' specifies how a compiler should implement the act of calling a procedure or function, concerning things like the order in which parameters are passed, where they are passed, and who is responsible (caller or callee?) for general housekeeping. Getting the convention right is crucial: fail to specify one when writing a dynamic library, and you may find only Delphi clients can call it; fail to specify one when writing the imports for a DLL or dylib, and dire things are likely to happen at runtime.

When targeting 64 bit Windows, deciding which calling convention to use is nevertheless easy, since Microsoft (sensibly enough) suggests using only one, the 'Microsoft x64 calling convention', and the Win64 Delphi compiler (equally sensibly) supports it and only it. In 32 bit land things are rather different though — the default convention in C is different to the default one in Delphi, which is different again to what Microsoft's C++ compiler (MSVC) uses as default for various things, which is different once more to the calling convention used by the Windows API!

Stemming from this, the 32 bit Delphi compiler supports a range of different conventions, specifically ‘`cdecl`’, ‘`stdcall`’, ‘`pascal`’ and ‘`register`’ (alias ‘fastcall’ or ‘Borland fastcall’). Any but for `register` (which is the default) is put into effect by suffixing the declaration of a procedure or function with a semi-colon and the name of the convention. On Windows you should use `stdcall`, and on OS X `cdecl`:

```
function AddThem(Num1, Num2: Int32): Int32;
  {$IFDEF MSWINDOWS}stdcall;{$ELSE}cdecl;{$ENDIF}
```

Once that has been done, the fact a convention other than Delphi’s own is being used is an implementation detail — the function will still be written or called as normal.

Please note that in this chapter, code snippets will not always specify the calling convention. This is just to keep things simple when explaining other matters however: in practice, the convention should *always* be specified.

Not letting exceptions escape

Since the way raising and handling errors differs between languages, it is important not to allow exceptions to ‘escape’ from a dynamic library. As a consequence, writing libraries is one of the few occasions when a ‘catch all’ try/except block can be a good idea. For example, each exported routine could be a function whose result indicates whether an exception happened or not:

```
function Foo: Boolean;
begin
  try
    {do stuff that may raise an exception}
    Result := True;
  except
    Result := False;
  end;
end;
```

On Windows, something very similar can be automated by using the `safecall` calling convention:

```
procedure Foo; safecall;
begin
  {do stuff that may raise an exception}
end;
```

As compiled, this comes to the exactly same thing as the following:

```
function Foo: HRESULT; stdcall; //an HRESULT is an integer
begin
  try
    {do stuff that may raise an exception}
    Result := S_OK;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

Other languages may or may not support an analogue to the `safecall` directive though — in particular, while C# and other .NET languages do (we will look at how exactly shortly), Visual Basic for Applications does not. If a target language does not support it, then the import should reflect the `stdcall` equivalent. For example, assuming `Foo` is compiled into a DLL called `MyDelphiLib.dll`, a VBA declare statement for it should look like the following:

```
Declare Function Foo Lib "MyDelphiLib.dll" () As Long
```

Using C compatible types

If you want a dynamic library to be useable by the broadest range of clients, both parameter and return types of exported routines should be C compatible. This means sticking to simple and pointer types, together with static arrays and records that only contain simple types, and avoiding dynamic arrays, classes and strings.

Naturally, there will be times when you really want to return either a dynamic array or string. On such an occasion, one good approach is for library to implement a pair of functions, one returning the length of the array or string and the second returning a copy of its data. The receiver then allocates a buffer, a pointer to which together with its length being passed in. The buffer is then filled by the library, with the number of elements actually copied returned as the function result.

For example, imagine a library that would ideally return a dynamic array of integers. Using C-compatible types, it might be written like this:

```
var
```

```
  InternalNums: TArray<Integer>;

function GetNumsLength: Integer; stdcall;
begin
  Result := Length(InternalNums);
end;

function GetNums(Buffer: PInteger;
  MaxNums: Integer): Integer; stdcall;
begin
  if (Buffer = nil) or (MaxNums <= 0) or (InternalNums = nil) then
    Exit(0);
  Result := Length(InternalNums);
  if Result > MaxNums then Result := MaxNums;
  Move(InternalNums[0], Buffer^, SizeOf(InternalNums[0]) * Result);
end;
```

A Delphi client can then retrieve the data as thus:

```
function GetNumsArray: TArray<Integer>;
begin
  SetLength(Result, GetNumLength);
  if Result <> nil then GetNums(@Result[0], Length(Result));
end;
```

A VBA client could act similarly:

```
Declare Function GetNumLength Lib "MyDelphiLib.dll" () As Long
Declare Function GetNums Lib "MyDelphiLib.dll" _
  (BufferPtr As Long, MaxNums As Long) As Long

Function GetNumsArray() As Long() ' a VBA Integer is 16 bits wide
  Dim Count As Long, Nums() As Long
  Count = GetNumLength
  If Count > 0 Then
    ReDim Nums(0 To Count - 1);
    GetNums VarPtr(Nums(0)), Count;
  End If
  GetNumsArray = Nums
End Function
```

For going the other way — i.e., to send an array of integers to a DLL — make the DLL routine take two parameters, one a pointer to the first element of the source array and the other a value that specifies the number of elements it holds. The data passed in can then be copied to a local buffer on the DLL side:

```
var
  InternalArray: TArray<Integer>;

procedure SetNums(Buffer: PInteger; Count: Integer); stdcall;
begin
  if (Count <= 0) or (Buffer = nil) then
    InternalArray := nil
  else
  begin
    SetLength(InternalArray, Count);
    Move(Buffer^, InternalArray[0], Count * SizeOf(Buffer^));
  end;
end;
```

The user of the library can then send over an array (static, dynamic or open) by referencing its first element:

```
procedure SendNumsToLibrary(const ANums: array of Integer);
begin
  if Length(Nums) = 0 then
    SetNums(nil, 0)
  else
    SetNums(@Nums[0], Length(Nums));
end;
```

Or, using VBA:

```
Declare Sub SetNums Lib "MyDelphiLib.dll" _
  (BufferPtr As Long, Count As Long)

Sub SendNumsToLibrary(Nums() As Long)
  Dim MinIndex As Long, MaxIndex As Long
  MinIndex = LBound(Nums)
  MaxIndex = UBound(Nums)
  If MaxIndex < MinIndex Then
```

```
    SetNums(0, 0)
  Else
    SetNums(VarPtr(Nums(MinIndex)), MaxIndex - MinIndex + 1)
  End If
End Sub
```

Passing strings using PChar

In the case of string data, the C-compatible alternative to the `string` type is to use either `PAnsiChar` or `PWideChar`/`PChar`, preferably the latter, though a VBA client will need `PAnsiChar` unless carefully implemented (VBA uses UTF-16 internally like Delphi, but assumes it will only call out to 'Ansi' DLLs). Since C-style strings have their lengths specified by a 'null terminator', things are slightly different to the dynamic arrays case. The library side might look like this:

```
var
  Description: string;

function GetDescriptionLength: Integer; stdcall;
begin
  Result := Length(Description);
end;

function GetDescription(Buffer: PWideChar;
  BufferLen: Integer): Integer; stdcall;
begin
  if (Buffer = nil) or (BufferLen = 0) then Exit(0);
  Result := Length(Description);
  if Result >= BufferLen then Result := BufferLen - 1;
  StrLPCopy(Buffer, Description, Result);
end;

procedure SetDescription(Desc: PWideChar); stdcall;
begin
  Description := Descr;
begin

exports
  GetDescription, GetDescriptionLength, SetDescription;
```

This code follows the prudent convention used by the traditional Windows API, in which the buffer length passed in *includes* the single element needed for the null terminator, but the return value *excludes* it. Calling the DLL from Delphi would then look like the following:

```
procedure ChangeDescription(const NewValue: string;
  out OldValue: string);
begin
  SetLength(OldValue, GetDescriptionLength);
  GetDescription(PWideChar(OldValue), Length(OldValue));
  SetDescription(PWideChar(NewValue));
end;
```

A VBA client would be able to set the string using the `StrPtr` 'magic' function:

```
Declare Sub SetDescription Lib "MyDelphiLib.dll" _
  (BufferPtr As Long)

Sub ChangeDescription(NewValue: String)
  SetDescription(StrPtr(NewValue));
End Sub
```

A C, C++, C# or even VB.NET client would be able to read and write in a similar fashion to a Delphi client however.

Using system-defined types

In a platform-specific way, an alternative to using just straight C types is to utilise system-allocated ones as well. On OS X, this might mean using Core Foundation objects as intermediaries (`CFString`, `CFData`, etc.). On Windows, it means Delphi's COM types can be used, in particular `WideString` and `OleVariant`, albeit with one caveat: due to the slightly different way Embarcadero and Microsoft compilers understand how a `stdcall` function result should be implemented, neither `WideString` nor `OleVariant` should not appear as a `stdcall` function result type. Instead, use either `safecall` or an `out` parameter:

```
function Wrong: OleVariant; stdcall;
function OK: OleVariant; safecall;
procedure AlsoOK(out S: OleVariant); stdcall;
```

If you need to write DLLs for VBA clients, using `OleVariant` (just `Variant` on the VBA side) can be particularly convenient, given the hassles of passing strings to and forth without VBA's anachronistic auto-conversions to and from 'Ansi' getting in the way. Unfortunately, these auto-conversions makes using `WideString` difficult, despite the thing it wraps — the COM BSTR — being VBA's native string type!

Nonetheless, `WideString` works great for passing strings to and from more capable environments such as .NET. For example, consider the following Delphi export:

```
function GetDescription: WideString; safecall;
```

If this function is exported by `MyDelphiLib.dll`, the P/Invoke declaration using C# syntax looks like this (the VB.NET syntax would be similar):

```
[DllImport("MyDelphiLib.dll", PreserveSig = false)]
[return: MarshalAs(UnmanagedType.BStr)]
static extern string GetDescription();
```

Setting the `PreserveSig` attribute set to `False` is the C# equivalent of using `safecall` in Delphi; leave it off if you want a `stdcall` function that returns an `HRESULT` instead:

```
[DllImport("MyDelphiLib.dll")]
static extern int Right(MarshalAs(UnmanagedType.BStr)]ref string S);
```

Due to the fact a Delphi `IInterface` is the same as a COM `IUnknown`, it is also possible to expose interfaces consumable by a C++ or .NET application without writing a full COM/ActiveX library, so long as the interface methods adhere to the interoperability rules for standalone routines. For example, consider the following Delphi DLL:

```
library DelphiInterfaceTest;

type
  ICalculator = interface
  ['{B1909D19-8DF8-4A4F-AF71-D0EFC653912F}']
    function AddThem(Num1, Num2: Int32): Int32; safecall;
  end;

  TCalculator = class(TInterfacedObject, ICalculator)
    function AddThem(Num1, Num2: Int32): Int32; safecall;
  end;

function TCalculator.AddThem(Num1, Num2: Int32): Int32;
begin
  Result := Num1 + Num2;
end;

function CreateCalculator: ICalculator; safecall;
begin
  Result := TCalculator.Create;
end;

exports
  CreateCalculator;
end.
```

Imports for the `CreateCalculator` function and `ICalculator` interface look like this in C#:

```
[Guid("B1909D19-8DF8-4A4F-AF71-D0EFC653912F")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
interface ICalculator
{
  Int32 AddThem(Int32 Num1, Int32 Num2);
}

class DelphiLib
{
  [DllImport("DelphiInterfaceTest.dll", PreserveSig = false)]
  public static extern ICalculator CreateCalculator();
}
```

The GUID in the C# declaration must match the GUID in the Delphi one, and the methods must be listed in the same order. The `InterfaceType` attribute must also be set to say it is an `IUnknown`-based interface. The actual method declarations can be a bit simpler than in the standalone routine case though, since the .NET marshaller now defaults to a `BSTR` mapping for `string` and `PreserveSig = false`.

In general terms, it can nonetheless be prudent to err on the side of caution and always use explicit `MarshalAs` attributes when writing P/Invoke imports. This is because P/Invoke marshalling defaults are not always intuitive from a Delphi point of view. For example, imagine wanting to map a `Boolean` parameter on the Delphi side to the appropriate C# type.

On the face of it you should be able to use `bool` and be done with it, given a C# `bool` and a Delphi `Boolean` are identical (both are one byte in size, and give `False` an ordinal value of 0 and `True` an ordinal value of 1). However, in a P/Invoke declaration, the .NET marshaller defaults to understanding `bool` to mean a Windows API `BOOL`, which is 4 bytes long. As a consequence, a `MarshalAs` attribute must be used to override this assumption:

```
function TestFlag: Boolean; stdcall;

//wrong!
[DllImport("MyDelphiLib.dll")]
static extern bool TestFlag();

//right!
[DllImport("MyDelphiLib.dll")]
[return: MarshalAs(UnmanagedType.I1)]
static extern bool TestFlag();
```

Linking to custom libraries

In order for a Delphi application to use a custom library, import declarations for it will need to be written. These take two main forms, producing load-time and runtime dynamic linking respectively. Sometimes the distinction is referred to as 'static' vs. 'dynamic' linking, however that can get confusing ('statically linking' a 'dynamic link' library...?).

In practice, the difference between load-time and runtime linking is this: where linking at runtime means the library is only loaded (typically explicitly) once the program is running, load-time linking has the operating system do the deed when the application starts up. When either an imported routine or whole library is not found, load-time linking will lead the operating system to show an error message, and not let your program run at all. This makes it less flexible than runtime linking. However, it is simpler to code, and allows for easily determining what a program's dependencies are with free tools such as Dependency Walker (depends.exe) on Windows and MacDependency on OS X.

Implementing load-time linking

In the case of a library written in Delphi, load-time linking requires import declarations that closely mirror the declarations of the original routines — the only difference is the addition of an `external` directive that specifies the name of the DLL or dylib the routine is implemented in. For example, given the following export:

```
function AddThem(Num1, Num2: Int32): Int32;

exports
  AddThem;
```

the corresponding import declaration on Windows would look like this:

```
function AddThem(Num1, Num2: Int32): Int32;
  external 'MyLibrary.dll';
```

On the Mac, you would just amend the library name accordingly:

```
function AddThem(Num1, Num2: Int32): Int32;
  external 'libMyLibrary.dylib';
```

Or, combined in with an IFDEF:

```
const
  LibName = {$IFDEF MSWINDOWS}'MyLibrary.dll'
            {$ELSE}'libMyLibrary.dylib'{$ENDIF};

function AddThem(Num1, Num2: Int32): Int32; external LibName;
```

As a minimum you must specify the library's name, though you can if you want hard code its path too. Usually that isn't done though, in which case the DLL or dylib can be placed in the same directory as your program in order to be found. (The exception is dylibs installed as standard on OS X, in which case the full path typically *is* specified, for example '`/usr/lib/libsqlite3.dylib`' for the Sqlite v3 library.)

In both the previous examples, `AddThem` was imported using its exported name (i.e., `AddThem`). You can choose another name if you prefer however:

```
function AddNums(Num1, Num2: Int32): Int32;
  external LibName name 'AddThem';
```

Here, `AddThem` would appear as `AddNums` to code in the Delphi client program.

Implementing runtime linking

On Windows, the code required for runtime linking itself takes two forms. In the first, you declare the import as previously shown, only with a `delayed` directive added at the end (notice the lack of a semi-colon before the `delayed` keyword):

```
function AddThem(Num1, Num2: Int32): Int32; external LibName delayed;
```

When the `delayed` directive is included, responsibility for loading the DLL passes to the RTL, which only attempts to do the deed the first time an imported routine is called. If you import a series of routines from the same library, using `delayed` on all of them will allow you to call one even if another doesn't exist: in other words, the `delayed` directive works at the routine level, not the library level.

Unfortunately, the exception that gets raised on failure is pretty uninformative — if there's any sort of problem in finding the routine, an `EExternalException` is raised with just a hexadecimal number for an error message. If you want something more helpful, even just to report what DLL or routine was missing, you need to plug in a custom callback — more on which shortly.

The second way to implement runtime linking on Windows — and the only way on the Mac — is to explicitly load the library, get the address to the required routine(s), then when you've finished with it, unload the library. This is done with three functions, `LoadLibrary`, `GetProcAddress` and `FreeLibrary`:

```
function LoadLibrary(LibraryName: PChar): HMODULE;
function GetProcAddress(Handle: HMODULE; ProcName: PChar): Pointer;
function FreeLibrary(Handle: HMODULE): LongBool;
```

On Windows, these reside in the `Windows.Winapi` unit; for the Mac, `System.SysUtils` provides implementations that delegate to the equivalent POSIX API functions (`dlopen`, `dlsym` and `dlclose` respectively, declared in `Posix.Dlfcn`), which you can call directly instead if you prefer.

In use, `LoadLibrary` (or `dlopen`) returns a numeric handle to the library specified; on failure, 0 is returned. This handle needs to be stored, firstly to pass to `GetProcAddress` to retrieve the addresses of the routines you want to call, and finally to `FreeLibrary` when you've finished actually calling them. Usual practice is to call `LoadLibrary` once, get the addresses of all the routines needed, then call `FreeLibrary` only when your application exits, or at least, when you aren't expecting to use the library any more. In the case of a library that returns interface references, make sure any references still held are set to `nil` before unloading the library, or you are likely to cause a crash later on.

`GetProcAddress` returns an untyped pointer that should be assigned to a variable typed appropriately. The procedural type in question will need to have been previously declared. In the case of our `AddThem` example, it could look like this (the names are all arbitrary):

```
type
  TAddNumsFunc = function (Num1, Num2: Int32): Int32;
```

Be careful to get the signature just right, including the calling convention (in the example, the native Delphi convention is assumed). Since both DLLs and dylibs store only the names of exported routines, mismatched signatures will only come to light when weird crashes start happening at runtime.

If you pass a name to `GetProcAddress` that isn't in the library's export table, `nil` will be returned. Check for this by first dereferencing the return value with the @ sign:

```
var
  AddNums: TAddNumsFunc;
  Lib: HMODULE;
begin
  Lib := LoadLibrary('MyLibrary.dll');
  if Lib = 0 then
  begin
    WriteLn('Could not load library');
    Exit;
  end;
  try
    AddNums := GetProcAddress(Lib, 'AddThem');
    if @AddNums = nil then
    begin
      WriteLn('Could not find exported routine');
      Exit;
    end;
    //call the function
    WriteLn('3 + 4 = ', AddNums(3, 4));
  finally
    FreeLibrary(Lib);
  end;
end.
```

Customising the behaviour of the 'delayed' directive

On Windows, using the `delayed` directive is perhaps the optimal way of importing DLL routines, since it combines the simplicity of load-time imports with the flexibility of traditional runtime ones. Nonetheless, and as previously noted, the standard behaviour when there's a problem isn't very user-friendly. To customise it, you need to write a special function to plug into the RTL. This must have the following signature (the name is arbitrary):

```
function DelayedCallback(dliNotify: dliNotification;
  pdli: PDelayLoadInfo): Pointer; stdcall;
```

If your aim is just to provide a more descriptive error message, implement the callback by checking for `dliNotify` equalling `dliFailLoadLibrary` or `dliFailGetProcAddress`. If it does, raise an exception of your choice; if it doesn't, return `nil`. To install it, the callback then needs to be registered with a call to `SetDliFailureHook`, usually in the initialization section of the callback's unit:

```
function DelayedCallback(dliNotify: dliNotification;
  pdli: PDelayLoadInfo): Pointer; stdcall;
begin
  Result := nil;
  case dliNotify of
    dliFailLoadLibrary:
      raise EDelayedLibraryLoadError.Create(pdli.szDll);
    dliFailGetProcAddress:
      if pdli.dlp.fImportByName then
      raise EGetProcAddressError.Create(pdli.szDll,
          pdli.dlp.szProcName)
    else
        raise EGetProcAddressError.Create(pdli.szDll,
          '#' + IntToStr(pdli.dlp.dwOrdinal));
  end;
end;

initialization
  SetDliFailureHook(DelayLoadFailureHook);
end.
```

The exception types themselves might be implemented like this:

```
interface

uses
  System.SysUtils;

type
  EDelayedLibraryLoadError = class(Exception)
  strict protected
    FLibraryName: string;
  public
    constructor Create(const AName: string);
    property LibraryName: string read FLibraryName;
  end;

  EGetProcAddressError = class(EDelayedLibraryLoadError)
  strict private
    FImportName: string;
  public
    constructor Create(const ALibraryName, AImportName: string);
    property ImportName: string read FImportName;
  end;

implementation

resourcestring
  SCannotLoadLibrary = 'Unable to load library (%s)';
  SGetProcAddressFail = 'Routine "%s" not found in library "%s"';

constructor EDelayedLibraryLoadError.Create(const AName: string);
begin
  inherited CreateFmt(SCannotLoadLibrary, [AName]);
  FLibraryName := AName;
end;

constructor EGetProcAddressError.Create(const ALibraryName,
  AImportName: string);
begin
  inherited CreateFmt(SGetProcAddressFail, [AImportName, ALibraryName]);
  FLibraryName := ALibraryName;
  FImportName := AImportName;
end;
```

Support routines (SafeLoadLibrary, GetModuleHandle, GetModuleName)

Two variants of `LoadLibrary` are `GetModuleHandle` and `SafeLoadLibrary`. The first returns a handle to an already loaded library, and 0 if the library hasn't already been loaded; since it doesn't do any actual loading itself, successful calls to `GetModuleHandle` are not paired with ones to `FreeLibrary`. `SafeLoadLibrary`, in contrast, wraps `LoadLibrary` to save and restore the control word of the floating point unit (FPU). This protects it from changes made by the library as it loads, which is something many Microsoft DLLs do — you'll find this has happened if after loading a DLL, Delphi's usual behaviour when invalid floating point calculations are made is messed up.

The final library-related function of note is `GetModuleName`. Declared in `System.SysUtils`, this returns the file name of a loaded library (if the handle passed in is invalid, an empty string is returned). The library can have been loaded either explicitly via `LoadLibrary`, or implicitly via an import declaration. The file name returned is the real, fully qualified name, which may not necessarily be the same as the name you passed in yourself, even if that was fully qualified too:

```
uses
  System.SysUtils, Macapi.Foundation;

const
  FwkPath = '/System/Library/Frameworks/' +
    'Foundation.framework/Foundation';
var
  Lib: HMODULE;
begin
  WriteLn('Loading the Cocoa Foundation framework:');
  WriteLn(FwkPath);
  WriteLn;
  Lib := LoadLibrary(FwkPath);
  WriteLn('Actual path:');
  WriteLn(GetModuleName(Lib));
  FreeLibrary(Lib);
end.
```

On my Mac, this code shows the 'real' path to the Cocoa framework being `/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation`, i.e. a couple of levels below the path specified when loading it.

Dynamic library search paths

Regardless of whether load-time or runtime linking is used, the usual case is not to specify the path to a dynamic library when linking to it. This leads to the system-specific DLL/dylib search path coming into play. On both Windows and OS X, the directory of the executable itself comes at the top of the places looked in. Beyond that the two operating systems diverge though, with Windows being more liberal and OS X much more restrictive.

In the case of Windows Vista and later, the DLL loader looks in the following places by default:

- The system directory for the bitness of the EXE (typically `C:\Windows\System32`, or `C:\Windows\SysWOW64` for a 32 bit EXE running on a 64 bit version of Windows).
- The 16 bit system directory (typically `C:\Windows\System`). A 64 bit EXE will *not* be able to find a DLL in the 32 bit system directory, just as a 32 bit EXE will not be able to find one in the 64 bit system directory.
- The Windows directory (typically `C:\Windows`).
- The current directory.
- The directories that are listed in the PATH environment variable.

Windows 2000 and Windows XP look in the same places, only with the current directory rising to the second place searched rather than the fifth.

OS X, in contrast, leaves relatively few places for dylibs to reside. As with Windows, the first place looked under is the executable's own directory. Beyond that, the following will typically be searched:

- `~/lib` (small 'l'; can be created if it doesn't already exist)
- `/usr/lib` (contains lots of pre-built open source dylibs by default)
- `/usr/local/lib`

If using load-time linking and you intend certain dylibs to reside near to, but not actually *in* the executable's own directory, use the Mac-specific `'@executable_path'` token as a placeholder inside a relative path specification:

```
const
  LibName = '@executable_path/../Libraries/libMyUtils.dylib';

function AddThem(A, B: Int32): Int32; cdecl; external LibName
  name '_AddThem';
```

In this example, `libMyUtils.dylib` is expected to reside in a `Libraries` folder that sits beside the executable's own (two dots meaning 'go up a level').

Platform specific issues

Windows issues

Simply put, if you wish to call a DLL from a 32 bit Windows application, the DLL itself must be 32 bit. Likewise, 64 bit EXEs must be matched with 64 bit DLLs. The only mixed setup supported by the operating system is when the library takes the form of an out-of-process COM server, in which it won't technically be a 'library' any more but another EXE.

A second Windows-specific thing to watch out for is the fact the current directory is in the DLL search path. Since an application has only some control over what its current directory is, this is usually considered a security risk. While your program can call `SetCurrentDir` as it wishes, the initial value of the current directory is user configurable, and moreover, common dialog boxes are liable to change it at will.

Nonetheless, the current directory can be removed from the search list by calling the `SetDllDirectory` API function when your program starts up, passing it an empty string:

```
SetDllDirectory('');
```

This function is declared in the `Winapi.Windows` unit, and is available in XP Service Pack 1 and later. If your application may be running on Windows 2000 too, you will need to go round the houses a bit instead:

```
procedure CleanDllSearchPath;
var
  Func: function (lpPathName: PWideChar): BOOL; stdcall;
begin
  Func := GetProcAddress(GetModuleHandle(kernel32),
    'SetDllDirectoryW');
  if @Func <> nil then Func('');
end;
```

Since the kernel will always be loaded, a single `GetModuleHandle` call can be used instead of a `LoadLibary`/`FreeLibrary` pair.

Mac-specific issues: the preceding underscore problem

While OS X doesn't have the current directory problem, it does have a rather silly underscore one instead. In short, while it is perfectly possible for a C-compatible, cdecl-using export to be named without a leading underscore, OS X's implementation of `dlsym` (the POSIX API function for getting the address of an library routine) 'helpfully' prepends an underscore to whatever name you pass it. If you wish to allow using runtime linking with a dynamic library you write, you therefore *must* export what you do with leading underscores attached. However, when using load-time linking in a Delphi client, the import declaration must then take account of the leading underscores so added.

If we go back to our `AddThem` example, as exported it should look like the following:

```
function AddThem(Num1, Num2: Int32): Int32;
  {$IFDEF MSWINDOWS}stdcall{$ELSE}cdecl{$ENDIF};
begin
  Result := Num1 + Num2;
end;

exports
  AddThem {$IFDEF MACOS} name '_AddThem';
```

The static import declaration for a Delphi client would then look like this:

```
const
  LibName = {$IFDEF MACOS}'libMyLibrary.dylib'
            {$ELSE}'MyLibrary.dll'{$ENDIF};

function AddThem(Num1, Num2: Int32): Int32;
  {$IFDEF MSWINDOWS}stdcall{$ELSE}cdecl{$ENDIF}; external LibName;
  {$IFDEF MACOS}name _AddThem;{$ENDIF}
```

Code for runtime linking should however forget about the leading underscore:

```
type
  TAddThemFunc = function AddThem(Num1, Num2: Int32): Int32;
    {$IFDEF MSWINDOWS}stdcall{$ELSE}cdecl{$ENDIF};

var
  Lib: HMODULE;
  AddThemFunc: TAddThemFunc;
begin
  Lib := LoadLibrary(LibName);
  If Lib = 0 then raise Exception.Create('Library not found');
  AddThemFunc := GetProcAddress(Lib, 'AddThem');
```

Dealing with flat functions that return Objective-C objects

Another Mac-specific issue is what to do when an exported routine uses Objective-C classes amongst its parameter or return types. For example, the Cocoa Foundation Framework exports a set of utility routines that return instances of Cocoa classes such as `NSArray` and `NSString`. Try to look for import declarations of these routines in the standard `Macapi.*` units and you will be looking in vain though. The reason for this is that the Delphi to Objective-C bridge only works in the context of wrapping Cocoa classes and metaclasses, not freestanding routines.

For example, one such Foundation function is `NSUserName`. This has the following declaration in Objective-C:

```
NSString * NSUserName (void);
```

Naïvely, you might think it should be translated like this:

```
function NSUserName: NSString; cdecl;
```

However, that won't work, since `NSString` on the Delphi side is a wrapper interface type, and in the case of a flat export, the Delphi RTL won't be able to trap any call to `NSUserName` to actually instantiate the wrapper.

Nevertheless, these sorts of export can still be used if you type the result either to `Pointer`, using `TNSString.Wrap` to manually wrap the return value as necessary, or type to the toll-free bridged Core Foundation equivalent. In the case of `NSString` this is `CFStringRef`, for a `NSArray` return type it would be `CFArrayRef` and so on:

```
uses
  Macapi.CoreFoundation, Macapi.Foundation;

const
  Fwk = '/System/Library/Frameworks/Foundation.framework/Foundation';

function NSUserName: CFStringRef; cdecl;
  external Fwk name '_NSUserName';

function GetUserNameCocoa: string;
var
  R: CFRange;
  S: CFStringRef;
begin
  S := NSUserName;
  R.location := 0;
  R.length := CFStringGetLength(S);
  SetLength(Result, R.length);
  CFStringGetCharacters(S, R, PChar(Result));
end;
```

FireMonkey issues

Traditionally, a popular reason for writing DLLs in Delphi was to expose VCL forms to applications written in other environments, or even just applications written in another version of Delphi. Since it is neither safe nor (for non-Delphi clients) practical to consume Delphi classes directly, the basic technique is to export a function that creates the form, shows it modally, then frees it:

```
library MySuperDialogLib;

uses
  System.SysUtils, System.Classes, Winapi.Windows, Vcl.Forms,
  MySuperDialog in 'MySuperDialog.pas' {frmMySuperDialog};

function ShowMySuperDialog(AOwner: HWND;
  ACaption: PWideChar): Integer; stdcall;
var
  Form: TForm;
begin
  Application.Handle := AOwner;
  Form := TfrmMySuperDialog.Create(nil);
  try
    Form.Caption := ACaption;
    Result := Form.ShowModal;
  finally
    Form.Free;
    Application.Handle := 0;
  end;
end;
```

```
exports
  ShowMySuperDialog;
end.
```

In principle, this code should need very little modification to expose a FireMonkey rather than a VCL form — delete `Winapi.Windows` from the `uses` clause, replace `Vcl.Forms` with `FMX.Forms`, and remove the `AOwner` parameter and setting of `Application.Handle` (in fact, setting `Application.Handle` isn't strictly necessary even in a VCL context). Unfortunately though, using FireMonkey forms in dynamic libraries involves hassles not found with VCL forms.

On Windows, these are caused by FireMonkey using GDI+, a Microsoft graphics library bundled with the operating system. Unlike the 'classic' GDI (Graphics Device Interface), GDI+ has some explicit 'start up' and 'shut down' requirements that the FireMonkey framework will perform for you in the context of an EXE, but won't in the context of a DLL.

In the first instance, this means the client application must call the `GdiplusStartup` and `GdiplusShutdown` API functions appropriately. When the host itself uses the FireMonkey framework, this will be done automatically. In the case of a VCL-based client, you can just use the `Winapi.GDIOBJ` unit, which calls `GdiplusStartup` in its `initialization` section and `GdiplusShutdown` in its `finalization` section; otherwise, the host may need to call these functions explicitly.

Beyond this, a further thing the client application must do is to use runtime not load-time linking to the FireMonkey DLL. In a Delphi context, this needs to be done using explicit `LoadLibrary`/`GetProcAddress`/`FreeLibrary` calls — simply using the `delayed` directive won't do. The reason stems from how GDI+ must be shut down only after all GDI+ objects have been freed — don't do this, and GDI+ is liable to crash the application at some point. For performance reasons, the FireMonkey framework will only free some of the GDI+ objects it creates when the framework itself is finalised though, which in the context of a DLL will be when the DLL is unloaded. As DLLs imported using the `delayed` directive are unloaded only when the program terminates, any `GdiplusShutdown` call (whether by yourself or left to `Winapi.GDIOBJ`) will be made too late.

Matters are even worse on OS X, though for different reasons: simply put, the way FireMonkey interacts with the Cocoa framework means a FireMonkey dylib only works well with a host application that is not a Cocoa application. Since FireMonkey on OS X is itself based on Cocoa, this makes normal FireMonkey dylibs incompatible with FireMonkey executables — instead, you must use packages, and as a result work with packages' usual requirement of both dylib and host using the same compiler and FireMonkey versions.

15. Multithreading

Multithreaded programming, or in other words, having different lines of code run at the same time, is a powerful technique to employ. Used appropriately, the fact even entry-level PCs nowadays have multicore processors means multithreading can deliver much improved performance, allowing tasks to be divided up into sub-tasks that get completed concurrently rather than one after the other. Even when the possibilities for true concurrency are limited, multithreading still has its place, being for example the standard solution to the problem of a user interface 'locking up' when a long-running operation is being performed, be this a complex calculation, a database being polled, files being downloaded from the internet, and so on.

The downside of multithreading is that it can be tricky to get right though. In contrast to concurrent 'processes' (applications or application instances), the various threads of the same process inhabit the same memory space. While this makes sharing data between threads very easy — which sounds a good thing — the truth of the matter is that it makes sharing *too* easy: quite simply, threads will step on each other's toes if you are not careful. As a result, if one thread attempts to read a group of values while another writes to them, the first thread will end up with data that is partly outdated and partly up-to-date. Due to the fact only read operations with certain basic types, and write operations pertaining to single byte values, are 'atomic' (guaranteed to complete without interference), this problem is liable to bite whenever data is shared between threads.

Aspects of multithreading

In general terms, multithreaded programming requires three areas of support from the development environment:

1. A place to put code that defines what a secondary thread is to do.
2. The availability of ‘synchronisation’ mechanisms to allow threads to safely share resources. Ideally, resources won’t be shared between running threads in the first place, e.g. by ensuring secondary threads create their own private copies of shared resources before they get going. However, sometimes it just can’t be helped — for example, calls to output status information to the screen will not be ‘thread-safe’ without explicit synchronisation. (The concept of ‘thread safety’ has various aspects to it, but in a nutshell, for a resource to be thread-safe means it won't get corrupted if multiple threads attempt to use it at the same time.)
3. The provision of ways for threads to safely notify each other when something of interest has happened, and conversely, wait on another thread to make a notification of this sort. For example, one thread may wish to notify another that data is now ready for processing, or that access to a scarce resource is now available, or that the user wishes to abort, and so on.

Delphi’s in-the-box multithreading support covers all three areas, and in a (mostly) cross platform manner:

- Idiomatically, defining a new thread means creating a `TThread` descendant.
- In itself, `TThread` has some basic synchronisation support for updating the GUI in a thread-safe manner. Beyond that, a range of primitives is available to be deployed, in particular ‘critical sections’ and ‘mutexes’, together with ‘spin locks’ and other more specialised primitives.
- For inter-thread notification, 'events', 'semaphores' and 'monitors' are the main tools to hand, along with thread-safe queues (lists) for the actual exchange of data.

The rest of this chapter will be structured around the three areas. In each case, we will begin with `TThread`, before branching out to other classes and functions; each section will then end with more specialised functionality.

Defining secondary threads

Every application will have at least one thread, namely the ‘main’ or ‘primary’ thread. This will be associated with the ‘process’ or application instance, beginning when the application starts up and ending when the application terminates.

Threads you define yourself will be secondary or ‘background’ threads. These can be run and terminated at will, though if any are still running when the main thread finishes, the operating system will just terminate them for you.

In principle, you can create a seemingly endless number of secondary threads, within overall system limits. If there are more threads running than processor cores however, the operating system will constantly flick back and forth between each thread, giving all a chance to run. This creates the illusion of concurrency even when threads aren’t really running at exactly the same time. Each ‘flick’ to another thread is called a ‘context switch’, and occurs at a very low level — in particular, you cannot expect the operating system to allow the code for a single Delphi procedure or even just a single Pascal statement to be completed before another thread is given a chance to run for a bit.

TThread basics

To create a secondary thread in Delphi, usual practice is to define a custom descendant of the `TThread` class, as declared in the `System.Classes` unit. When doing this, you *must* override the parameterless `Execute` method, which defines the code to be run inside the new thread:

```
type
  TExampleThread = class(TThread)
  //...
  protected
    procedure Execute; override;
  end;

procedure TExampleThread.Execute;
begin
  { code to be actually run inside the secondary thread here... }
end;
```

When the `Execute` method ends, the thread terminates. Once that happens, it is impossible to restart the thread — instead, a new thread needs to be created.

If you define a custom constructor, the code in it will run in the context of the calling thread. This is because the new thread only actually starts once the construction process has finished. Sometimes even that is too soon; if so, pass `True` for the `TThread` constructor’s optional `CreateSuspended` parameter (the default is `False`). This will leave the thread in a ‘suspended’ state post-construction, requiring an explicit call to `Start` to get going:

```
var
  SecThread: TExampleThread;
begin
  SecThread := TExampleThread.Create(True);
  //... do some more initialisation
  SecThread.Start;
```

Freeing a TThread

By default, a `TThread` descendant will need to be explicitly freed. However, by setting the `FreeOnTerminate` property to `True`, the thread object will free itself soon after `Execute` finishes. Here’s a simple example of the three things so far discussed — defining a `TThread` descendant, requiring an explicit `Start` call, and enabling automatic destruction — in practice:

```
uses
  System.SysUtils, System.Classes;

type
  TExampleThread = class(TThread)
  protected
    procedure Execute; override;
  end;

procedure TExampleThread.Execute;
begin
  WriteLn('   The secondary thread says hello');
  Sleep(1000);
  WriteLn('   The secondary thread says goodbye');
end;

var
  Thread: TExampleThread;
```

```
begin
  Thread := TExampleThread.Create(True); //create suspended
  Thread.FreeOnTerminate := True;
  WriteLn('About to run the secondary thread...');
  Thread.Start;
  Sleep(500);
  WriteLn('It has probably said ''hi'' by now...');
  Sleep(2000);
  WriteLn('And it''s goodbye from the main thread too');
  ReadLn;
end.
```

Calling `Sleep` is just a proxy for 'doing actual work (the figure passed to it denotes the number of milliseconds to temporarily suspend the thread for). The numbers used should ensure the following output:

```
About to run the secondary thread...
   The secondary thread says hello
It has probably said 'hi' by now...
   The secondary thread says goodbye
And it's goodbye from the main thread too
```

TThread properties

While a thread is executing, the `ThreadID` property of the object will return a number that uniquely identifies it across the system. On Windows, this is different from the `Handle` property, which is process-specific but something that can be passed to API functions such as `WaitForMultipleObjects`. (There is no `Handle` on OS X, because POSIX threads only have IDs.) For debugging purposes, you can also 'name' threads; do this by calling `NameThreadForDebugging` at the top of your `Execute` method:

```
procedure TExampleThread.Execute;
begin
  NameThreadForDebugging('My super thread');
  //do stuff...
```

This will only be meaningful if you have the Professional edition or above — while the Starter edition includes the core parts of the RAD Studio debugger, it does not include the 'Threads' debug view specifically (in higher editions, go to `View|Debug Windows|Threads`, or press `Ctrl+Alt+T`, to access this).

If an exception is raised while a secondary thread is running and it is not explicitly handled, the thread will abort. At the operating system level, failing to handle an exception raised in a secondary thread will bring down the whole application. Because of this, the internals of `TThread` provide a default exception handler for you, capturing any exception not handled explicitly in a `try`/`except` block and assigning it to the `FatalException` property (the exception object will then get freed by the `TThread` destructor).

To make use of this, `TThread` provides an `OnTerminate` event property that can be assigned by the creator when a thread object is being created. How it works is like this: when a thread has finished executing, either cleanly or because an exception was raised, its `Finished` property will return `True`. Immediately before this property changes its value from `False`, but after `Execute` has completed, any `OnTerminate` handler will be called. At this point the secondary thread (and not just the thread *object*) will technically still exist. However, `OnTerminate` will be raised in the context of the main thread, so you can pretty much pretend the secondary thread *has* terminated at that point, leaving only the carcass of the thread object.

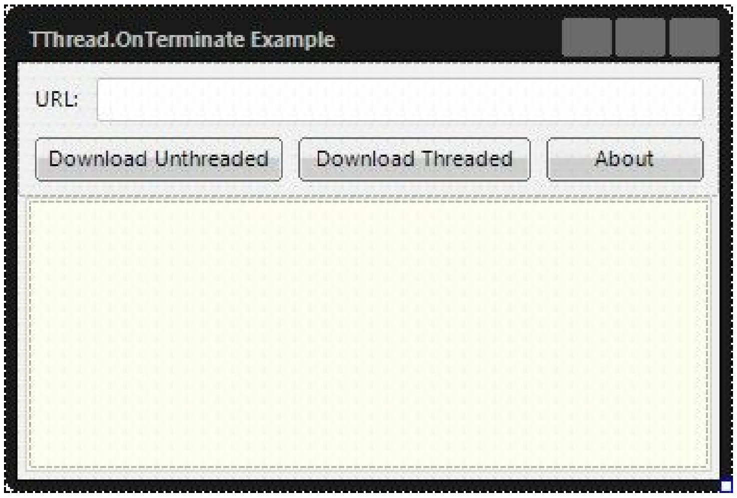
This may well sound a little abstract, so let's have an example. Bundled with Delphi is the open source 'Internet Direct' component suite for socket programming ('Indy' for short). This has a 'blocking' architecture, which means when you ask it do something, it doesn't return until it has done that, 'blocking' the caller from doing anything else in the meantime. One of the very many things it can do is download files from the internet, so we'll create a simple application that downloads something in the background, keeping the user interface responsive.

OnTerminate in practice

To download a file with Indy, create a `TIdHTTP` component (this class is declared in the `IdHttp` unit), and call its `Get` method. This takes several overloads, one of which receives a `TStream` to output to. Using it, we can download a JPEG image to a temporary memory stream and then send the data on to a `TImage` control for display.

If Indy is installed in the IDE (it will be by default), `TIdHTTP` will be available in the Tool Palette. Since it is best to limit sharing things between threads as much as possible though, we will create the component dynamically instead.

So, create a new FireMonkey HD application (a VCL version would be almost the same), and put a `TEdit` across the top of the form, together with three buttons captioned 'Download Unthreaded', 'Download Threaded' and 'About'. Below them, place a `TImage` for output:



Name the controls `edtURL`, `btnUnthreadedDL`, `btnThreadedDL`, `btnAbout`, and `imgOutput`. For the first bit of actual code, we'll handle the 'About' button's `OnClick` event. This can do anything really— its only purpose will be to demonstrate how the user interface remains responsive when a secondary thread is used. In our own case, let's just show a message box:

```
procedure TfrmDownloadTest.btnAboutClick(Sender: TObject);
begin
  ShowMessage('I''m definitely still alive!');
end;
```

Next we'll implement the unthreaded download. For this, add `IdHTTP` to the form's `uses` clause (either `interface` or `implementation` section will do), before handling the first button's `OnClick` event like this:

```
procedure TfrmDownloadTest.btnUnthreadedDLClick(Sender: TObject);
var
  IdHttp: TIdHTTP;
  Stream: TMemoryStream;
begin
  if edtURL.Text = '' then Exit;
  Stream := nil;
  IdHttp := TIdHTTP.Create(nil);
  try
    Stream := TMemoryStream.Create;
    IdHttp.Get(edtURL.Text, Stream);
    Stream.Position := 0;
    imgOutput.Bitmap.LoadFromStream(Stream);
  finally
    IdHttp.Free;
    Stream.Free;
  end;
end;
```

If you now run the application, enter a URL to a suitably large image file in the edit box before clicking the first button, the user interface will be frozen while the file downloads (if you don't have a suitable URL handy, I have put a 8MB PNG at `http://delphihaven.files.wordpress.com/2012/01/dublin.png` — if you have a fast connection that would eat such a thing for breakfast, then you'll need to find your own picture).

Now let's implement the threaded download. For this, we need to define a `TThread` descendant:

```
type
  TDownloadThread = class(TThread)
  strict private
    FData: TCustomMemoryStream;
    FIdHttp: TIdHTTP;
    FURL: string;
  protected
    procedure Execute; override;
  public
    constructor Create(const AURL: string);
    destructor Destroy; override;
    property Data: TCustomMemoryStream read FData;
  end;
```

Without work on your part, you should *not* expect any public properties you define to be thread-safe. Thus, the Data property should *not* be accessed while the thread is running. What it stands for in our case is the output data: where it *will* be accessed, then, is an `OnTerminate` handler.

Before getting to that though, we need to implement the thread class itself:

```
constructor TDownloadThread.Create(const AURL: string);
begin
  inherited Create(True); //create suspended
  FData := TMemoryStream.Create;
  FIdHttp := TIdHTTP.Create(nil);
  FURL := AURL;
end;

destructor TDownloadThread.Destroy;
begin
  FIdHTTP.Free;
  FData.Free;
  inherited;
end;

procedure TDownloadThread.Execute;
begin
  FIdHttp.Get(FURL, FData);
end;
```

Back in the form, `btnThreadedDL` can now have its `OnClick` event handled like this:

```
procedure TfrmDownloadTest.btnThreadedDLClick(Sender: TObject);
var
  Thread: TDownloadThread;
begin
  if edtURL.Text = '' then Exit;
  btnThreadedDL.Enabled := False;
  btnUnthreadedDL.Enabled := False;
  Thread := TDownloadThread.Create(edtURL.Text);
  Thread.FreeOnTerminate := True;
  Thread.OnTerminate := DownloadCompleted;
  Thread.Start;
end;
```

Declare `DownloadCompleted`, the thread object's `OnTerminate` handler, in the `private` or `strict private` section of the form:

```
type
  TfrmDownloadTest = class(TForm)
    imgOutput: TImage;
  //... etc.
  private
    procedure DownloadCompleted(Sender: TObject); //!!!add
```

It can be implemented as thus:

```
procedure TfrmDownloadTest.DownloadCompleted(Sender: TObject);
var
  Thread: TDownloadThread;
begin
  Thread := Sender as TDownloadThread;
  if Thread.FatalException <> nil then
    Application.HandleException(Thread.FatalException)
  else
    try
      Thread.Data.Position := 0;
      imgOutput.Bitmap.LoadFromStream(Thread.Data);
    except
      Application.HandleException(ExceptObject)
    end;
  lblInfo.Visible := False;
  btnThreadedDL.Enabled := True;
  btnUnthreadedDL.Enabled := True;
end;
```

Here, we first get a strongly-typed reference to the thread object, before checking whether it was aborted due to an exception. If it was, then we pass the exception object onto the application's default exception handler; if it wasn't, we read the data. The reading is wrapped in a `try`/`except` block which again passes on any exception to the default exception handler. Normally this passing on happens automatically in a Delphi GUI application, however it needs to be done explicitly when a secondary thread is involved.

If you rerun the application and choose the second button, you should now find the UI remains responsive while the thread downloads in the background. For example, the form can be moved and sized as normal, and the 'About' button can be clicked.

'Anonymous' threads

Instead of implementing your own `TThread` descendant, you can call `CreateAnonymousMethod`, a class method of `TThread`. This takes a parameterless anonymous method (procedure); internally, a special `TThread` descendant is created, which calls your anonymous method in its `Execute` override. The `TThread` object is created suspended (meaning you must explicitly call `Start` to get it going), and with `FreeOnTerminate` set to `True`:

```
uses IdHttp;

procedure StartDownloadThread(const URL: string; Dest: TCustomMemo);
var
  Thread: TThread;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    var
      IdHttp: TIdHTTP;
      HTML: string;
    begin
      IdHttp := TIdHTTP.Create(nil);
      try
        HTML := IdHttp.Get(URL);
      finally
        IdHttp.Free;
      end;
      TThread.Synchronize(TThread.CurrentThread,
        procedure
        begin
          Dest.Text := HTML;
        end);
    end);
  Thread.Start;
end;
```

In essence, `CreateAnonymousThread` is just a small shortcut, relieving you from the bother of defining a `TThread` descendant yourself.

Within the body of the anonymous method used, the underlying `TThread` instance can be retrieved by calling the `CurrentThread` class property of `TThread`. This in fact works for the main thread too, even though `TThread` doesn't really have a part to play in it:

```
ShowMessageFmt('The thread ID of the main thread is %d',
  [TThread.CurrentThread.ThreadID]);
```

In the jargon of `TThread` itself, any thread not created under its aegis is an 'external' thread:

```
if TThread.CurrentThread.ExternalThread then
  ShowMessage('I am running in a thread that ' +
    'was not started by a TThread instance')
else
  ShowMessage('I am running inside a 100% Delphi background thread');
```

Aside from the main thread, an 'external' thread might be one created by calling operating system API directly, or our next function to consider: `BeginThread`.

Low-level threading with BeginThread

Short of calling the operating system API directly, the lowest level you can create threads in Delphi is with the `BeginThread` function. This has a slightly different signature depending on whether you are targeting Windows or OS X:

```
type
  TThreadFunc = function(Parameter: Pointer): Integer;

//Windows version
function BeginThread(SecurityAttributes: Pointer;
  StackSize: LongWord; ThreadFunc: TThreadFunc;
  Parameter: Pointer; CreationFlags: LongWord;
  var ThreadId: TThreadID): THandle;

//OS X/POSIX version
type
  TThreadAttr = pthread_attr_t;
  PThreadAttr = ^TThreadAttr;

function BeginThread(Attribute: PThreadAttr;
  ThreadFunc: TThreadFunc; Parameter: Pointer;
  var ThreadId: TThreadID): Integer;
```

On Windows, `BeginThread` is a thin wrapper round the `CreateThread` API function; on OS X, it wraps the `pthread_create` POSIX function ('POSIX' is the standardised system API for Unix and Unix-type operating systems, of which OS X is an example). Where a return value of 0 means *failure* on Windows (anything else will be the 'handle' of the new thread), it means *success* on POSIX (non-zero values being error codes).

When calling `BeginThread`, pass `nil` and `0` to use the default OS-defined attributes and stack size respectively (this is the usual thing to do). `Parameter` can then be anything you want, and will just be passed on as the argument to `ThreadFunc` (pass `nil` if you have nothing to pass). On Windows, use `CREATE_SUSPENDED` for `CreationFlags` in order to have the new thread created in a blocking (not running) state (there is no such parameter for POSIX, since POSIX threads are always created running). When done, you must call the `ResumeThread` API function to have the code defined by `ThreadFunc` actually run. `ResumeThread` itself takes the thread handle, i.e. the value `BeginThread` returns; both it and `CREATE_SUSPENDED` are declared in `Winapi.Windows`.

Corresponding to `BeginThread` is `EndThread`:

```
procedure EndThread(ExitCode: Integer);
```

This may be called by the routine specified by `ThreadFunc` to terminate itself, however the thread will be terminated when that routine exits anyhow. In general, the value passed as the 'exit code' to `EndThread` (or returned by the thread function) isn't important. At the `TThread` level, it will be the final value of the `ReturnValue` property after the `Execute` method has finished, which will be 0 if `ReturnValue` was never explicitly set (this is the usual case).

After a thread created using `BeginThread` (or the API function it maps) has terminated, it (and the memory it has taken) will still exist in a zombie state until freed by another thread (typically the main thread) using an appropriate API function. On Windows this means calling `CloseHandle`, passing the value that `BeginThread` (or `CreateThread`) returned; on OSX, it means calling `pthread_detach` and passing the thread ID. `CloseHandle` is declared in `Winapi.Windows`, `pthread_detach` in `Posix.Pthread`.

Thread-local variables (threadvar)

Any local variables within `ThreadFunc` will be thread-specific. This in fact stands for the local variables of *any* routine or method. Beyond them, it is also possible to declare thread-specific global variables by using the `threadvar` keyword in place of the usual `var`. For example, consider the following thread function:

```
uses
  Winapi.Windows;

var
  SomeNumNormal: Integer;

threadvar
  SomeNumThread: Integer;

function ThreadFunc(Data: Pointer): Integer;
begin
  Inc(SomeNumNormal, 100);
  Inc(SomeNumThread, 100);
  WriteLn('Thread ', GetCurrentThreadID, ':');
  WriteLn('SomeNumNormal = ', SomeNumNormal);
  WriteLn('SomeNumThread = ', SomeNumThread);
  WriteLn('');
  Result := 0;
end;
```

From it, two threads are created:

```
var
  Handles: array[1..2] of THandle;
  IDs: array[1..2] of TThreadID;
  I: Integer;
begin
  SomeNumNormal := 42;
  SomeNumThread := 42;
  for I := Low(Handles) to High(Handles) do
  begin
    Handles[I] := BeginThread(nil, 0, ThreadFunc, nil, 0, IDs[I]);
    Sleep(100);
  end;
  WaitForMultipleObjects(Length(Handles), @Handles, True, INFINITE);
  for I := Low(Handles) to High(Handles) do
    CloseHandle(Handles[I]);
end.
```

When the program is run, output like the following will result:

```
Thread 928:
SomeNumNormal = 142
SomeNumThread = 100

Thread 3524:
SomeNumNormal = 242
SomeNumThread = 100
```

This shows that where `SomeNumNormal` is shared between the threads, each have their own copy of `SomeNumThread`.

Using `threadvar` does have its issues; in particular, using a managed type (e.g. `string`) will necessitate you explicitly finalising the variable, since that won't be done automatically like it would in the case of a normal variable or field. Thus, if you declare a `threadvar` typed to `string` it should be explicitly set to `''` when each thread exits; when typed to a dynamic array, interface or anonymous method, it should be set to `nil`; and when typed to a variant, it should be set to `Unassigned`.

All in all, both `threadvar` and `BeginThread` are rather specialised features. In general, using `TThread`, whether by implementing an explicit descendant class or by calling `CreateAnonymousThread`, is to be preferred.

Synchronising threads

Synchronising threads is not like synchronising watches: you don't synchronise threads so that they run at the same time. Rather, synchronised threads are ones that run *serially*, one after the other, or perhaps in turns.

On one level, *having* to serialise threads is always a defeat, since in an ideal world, no running thread would have to pause for another to do something first: instead, everything would just run concurrently. Unfortunately, synchronisation will usually be necessary whenever some sort of shared resource is written to. For example, the following code is likely to not work properly:

```
uses
  System.Classes;

var
  SecondaryThread: TThread;
begin
  SecondaryThread := TThread.CreateAnonymousThread(
    procedure
    begin
      WriteLn('Hello from the secondary thread');
    end);
  SecondaryThread.Start;
  WriteLn('Hello from the main thread');
  ReadLn;
end.
```

When I run this program myself, sometimes the main thread's message is outputted twice, sometimes never at all, and sometimes jumbled up with the secondary thread's message! This is because the console is a shared resource, and `WriteLn` is not wise to the possibility it may be called twice in succession when then first call has yet to complete.

What needs to be synchronised?

As a general rule of thumb, if a value read by one or more threads may be written to by another, then access to it must be synchronised. Since VCL and FireMonkey forms and controls are created in the main thread, this means changing the properties of any VCL or FireMonkey component is not thread-safe without explicit serialisation.

More fundamentally, it usually prudent to err on the side of caution when *any* objects are involved, unless you know for sure they are 'immutable' (i.e., not able to change their state once created). In contrast, the procedural RTL is generally thread-safe, though with some exceptions. We've already seen `WriteLn` fail the test; the other prominent examples are the string formatting routines (`Format`, `FloatToStr`, `DateTimeToStr`, etc.).

The problem with the formatting functions is that they make use of a global variable, `FormatSettings`. Nonetheless, each function is overloaded to take an explicit `TFormatSettings` record. If a secondary thread wishes to call `Format` or whatever, it should therefore construct its own `TFormatSettings` instance on being created, which can then be passed to any formatting function it calls in `Execute`:

```
type
  TMyThread = class(TThread)
  strict private
    FFormatSettings: TFormatSettings;
  protected
    procedure Execute; override;
  public
    constructor Create(ACreateSuspended: Boolean);
  end;

  constructor TMyThread.Create(ACreateSuspended: Boolean);
  begin
    inherited Create(ACreateSuspended);
    FFormatSettings := TFormatSettings.Create;
  end;

  procedure TMyThread.Execute;
  var
    S: string;
  begin
    //...
    S := DateToStr(Date, FFormatSettings);
    //...
  end;
```

At a more fundamental level still, the act of reading a string or dynamic array is thread-safe, notwithstanding the fact mutable reference counts are used internally, because the act of updating the reference count is serialised by the low-level RTL. However, something as basic as incrementing or decrementing an integer — `Inc(X)` and `Dec(X)` as much as `X := X + 1` and `X := X - 1` — is *not* thread-safe. This is because a thread may be interrupted in between getting the old value and assigning the new, leading 1 to be added to (or taken away from) an out of date version of `X`.

To say this may happen is to say the operation of incrementing or decrementing a variable is not an 'atomic' operation. On the face of it, this may seem a fundamental failing of Delphi. However, all mainstream languages suffer from it; in C and C-style languages, for example, the `++` and `--` operators suffer the same issue `Inc` and `Dec` do in Delphi.

Assuming the number in question is typed to either `Integer` or `Int64` however, there is a simple enough workaround in most cases, which is to use the `Increment`, `Decrement` or `Add` methods of `TInterlocked`. This is a static class declared in `System.SyncObjs`:

```
var
  X: Integer;
begin
  X := 0;
  TInterlocked.Increment(X); //add 1 to X atomically
  TInterlocked.Decrement(X); //subtract 1 from X atomically
  TInterlocked.Add(X, 22);   //add 22 to X atomically
  TInterlocked.Add(X, -11);  //subtract 11 from X atomically
```

Aside from changing the specified variable in place, these functions all return the new value too.

A limitation of `TInterlocked` is that it will only work correctly if the data being incremented or decremented is 'aligned'. This will usually be so unless you make it otherwise. The main way it might not is if the value being changed is a field of a 'packed' record:

```
type
  TBag = packed record
    First: Byte;
    Second: Integer;
  end;

var
  Backpack: TBag;
```

In this situation, `Backpack.Second` is not aligned because the `packed` directive forces it to come immediately after `Backpack.First` in memory. Since the default is to use natural alignment however, removing `packed` will make using `TInterlocked` on `Backpack.Second` safe.

TThread.Synchronize

In Delphi programming, probably the most common case for a secondary thread needing to serialise access to a shared resource is when it wishes to change the properties of a VCL or FireMonkey object, e.g. add a line to a memo control. For precisely this sort of scenario, `TThread` provides a simple solution in the form of the `Synchronize` method.

Passed a parameterless anonymous procedure, `Synchronize` suspends the calling thread while the method passed is performed in the main thread. In the following example, a secondary thread safely updates a form's background colour using `Synchronize`:

```
type
  TFormColorSwitcher = class(TThread)
  strict private
    FForm: TForm;
  protected
    procedure Execute; override;
  public
    constructor Create(AForm: TForm);
  end;

constructor TFormColorSwitcher.Create(AForm: TForm);
begin
  inherited Create;
  FForm := AForm;
  FForm.Fill.Kind := TBrushKind.bkSolid;
end;

procedure TFormColorSwitcher.Execute;

  function RandomElem: Byte;
  begin
    Result := Random(100) + 155;
  end;
var
  NewColor: TAlphaColor;
begin
  while not Terminated do //Terminated will be explained later
  begin
    Sleep(500);
    NewColor := MakeColor(RandomElem, RandomElem, RandomElem);
    Synchronize(
      procedure
      begin
        FForm.Fill.Color := NewColor;
      end);
  end;
end;
```

TThread.Queue

If two threads call `Synchronize` at the same time, then one of them will be suspended not just for the duration of its own delegate procedure being run, but the other thread's too. One way to avoid this possibility is to use `Queue` rather than `Synchronize`, `Queue` being another `TThread` method for simple synchronisation. As with `Synchronize`, you pass `Queue` an anonymous procedure that is then executed in the main thread. However, where `Synchronize` waits until this procedure has been performed, `Queue` does not. Instead, it posts the method to a queue (hence its name), which the main thread then periodically processes.

In general, using `Queue` should be preferred as much as possible to `Synchronize`, since the waiting behaviour of the latter can negate the point of using secondary threads in the first place. The main caveat to this advice concerns what happens when the secondary thread terminates before the main thread has got round to processing all its queued methods: in such a situation, the unprocessed methods will just be lost.

When you create an 'anonymous' thread, the thread procedure will not be able to call `TThread` instance methods. To cover this case, both `Synchronize` and `Queue` come in class as well as instance method guises:

```
  Thread := TThread.CreateAnonymousThread(
    procedure
    var
      I: Integer;
    begin
      for I := 1 to 10 do
        TThread.Queue(TThread.CurrentThread,
          procedure
          begin
            StatusBar.Text := 'Working' + StringOfChar('.', I);
            Sleep(50);
          end);
    end);
```

A final point to note about `Synchronize` and `Queue` is that they only work in GUI applications — in a console program, you will have to use some other technique.

Critical sections (TCriticalSection, TMonitor)

Perhaps the most common way to synchronise threads in Delphi is to use a 'critical section' object. Here, a thread requests to 'enter' the critical section; if another thread has got in first, then the second thread 'blocks' (waits its turn patiently). Once entered, the thread is said to have 'taken the lock'. Once it has finished doing what it needs to do with the shared resource, it leaves 'leaves' or 'exits' the critical section, releasing the lock and so allowing another thread to acquire it.

For historical reasons, there is more than one way to define a critical section in Delphi. The first is to use the `TMonitor` record, as declared in the `System` unit. Rather than declaring an instance of `TMonitor`, you use it to 'lock' an arbitrary class instance. For example, in order to achieve thread-safe writes to the console, you might implement a simple helper record that synchronises `WriteLn` calls like this:

```
type
  TConsole = record
  strict private
    class var FLock: TObject;
    class constructor Create;
    class destructor Destroy;
  public
    class procedure WriteLine(const S: string); static;
  end;

class constructor TConsole.Create;
begin
  FLock := TObject.Create; //a monitor 'enters' *any* object
end;

class destructor TConsole.Destroy;
begin
  FLock.Free;
end;

class procedure TConsole.WriteLine(const S: string);
begin
  TMonitor.Enter(FLock);
  try
    WriteLn(S);
  finally
    TMonitor.Exit(FLock);
  end;
end;
```

This can then be used like this:

```
var
  SecondaryThread: TThread;
begin
  SecondaryThread := TThread.CreateAnonymousThread(
    procedure
    begin
      TConsole.WriteLine('Hello from the secondary thread');
    end);
  SecondaryThread.Start;
  TConsole.WriteLine('Hello from the main thread');
  ReadLn;
end.
```

The second way to define a critical section is to use the `TCriticalSection` class, as declared in `System.SyncObjs`:

```
type
  TConsole = record
  strict private
    class var FCriticalSection: TCriticalSection;
    class constructor Create;
    class destructor Destroy;
  public
    class procedure WriteLine(const S: string); static;
  end;

class constructor TConsole.Create;
begin
  FCriticalSection := TCriticalSection.Create;
end;

class destructor TConsole.Destroy;
begin
  FCriticalSection.Free;
end;

class procedure TConsole.WriteLine(const S: string);
begin
  FCriticalSection.Enter;   //or FCriticalSection.Acquire
  try
    WriteLn(S);
  finally
    FCriticalSection.Leave; //or FCriticalSection.Release
  end;
end;
```

When targeting OS X, `TCriticalSection` is actually just a simple wrapper round `TMonitor`. In contrast, on Windows, it provides a (very) thin wrapper over the Windows API's critical section functions. On both platforms, `TMonitor` has a custom implementation though.

There are a few caveats to note regardless of which type you choose to use. The first is that the thread safety provided relies on each thread religiously calling `Enter` before it writes to the resource being protected. If a thread comes in and just ignores the critical section or monitor, nothing will stop it.

Secondly, when two threads act on a couple of shared resources, each of which have their own critical section guards, deadlocks can arise pretty easily if you are not careful: thread 1 requests and gets a lock on resource *A* as thread 2 requests and gets a lock on resource *B*; then, thread 1 requests a lock on resource *B*, followed by thread 2 requesting a lock on resource *A*. Neither of the second pair of `Enter` calls can return since each thread holds a lock on what the other one wants. Whoops!

As a partial workaround, `TMonitor` allows specifying a timeout value on calling `Enter` (the timeout is in milliseconds) — when the call times out, `False` rather than `True` is returned:

```
if not TMonitor.Enter(FLock, 5000) then //wait for 5 seconds
  //do something having timed out...
else
  try
    //work with guarded resource
  finally
    TMonitor.Exit(Flock);
  end;
```

However, this just allows recovering after a deadlock has happened, not preventing the deadlock from happening in the first place.

Mutexes (TMutex)

Mutexes in Delphi are essentially just a third form of critical section, and map directly to an operating system primitive on both Windows and OS X (POSIX). On Windows, a mutex is functionally distinguished from a normal critical section in being potentially cross-process and 'nameable'. This is at the cost of being slower than a normal critical section (or monitor) though. The interface is also slightly different: using `TMutex`, you 'acquire' or 'wait for' rather than 'enter', and 'release' rather than 'exit' or 'leave':

```
type
  TConsole = record
  strict private
    class var FMutex: TMutex;
    class constructor Create;
    class destructor Destroy;
  public
    class procedure WriteLine(const S: string); static;
  end;

class constructor TConsole.Create;
begin
  FMutex := TMutex.Create;
end;

class destructor TConsole.Destroy;
begin
  FMutex.Free;
end;

class procedure TConsole.WriteLine(const S: string);
begin
  FMutex.Acquire; //or FMutex.WaitFor(TimeoutInMSecs)
  try
    WriteLn(S);
  finally
    FMutex.Release;
  end;
end;
```

Mostly for Windows-specific things, the `TMutex` constructor has a few overloads:

```
constructor Create(UseCOMWait: Boolean = False); overload;
constructor Create(MutexAttributes: PSecurityAttributes;
  InitialOwner: Boolean; const Name: string;
  UseCOMWait: Boolean = False); overload;
constructor Create(DesiredAccess: LongWord;
  InheritHandle: Boolean; const Name: string;
  UseCOMWait: Boolean = False); overload;
```

`UseCOMWait` is only used in the context of Windows COM programming — internally, it means `CoWaitForMultipleHandles` rather than `WaitForMultipleObjectsEx` will be called by the `WaitFor` method. Of the other possible parameters, `MutexAttributes` can be a reference to a Windows API security attributes structure, otherwise pass `nil` (it will be ignored on OS X), `InitialOwner` determines whether the mutex should be acquired immediately, and `Name` is Windows only (passing anything other than an empty string for it will cause an exception to be raised on OS X).

Using mutexes

A popular use of mutexes on Windows is to only allow one instance of an application to run at any one time. If using `TMutex` rather than the Windows API directly, you need to do this:

- Create a mutex with an appropriate name, passing `False` for `InitialOwner`.
- Immediately attempt to acquire it, but don't block. This can be done by calling `WaitFor`, and passing 0 for the timeout. If successful, then great, you must be the first active instance, otherwise you can't be, so exit.
- Free the mutex on shutdown.

In itself, a mutex simply provides the mechanism for knowing whether an instance of the application is already running: it does not bring the first instance to the foreground, let alone pass on any command line parameters to it. For that you need to dip into the Windows API. Here is one implementation.

Firstly, create a new VCL application, and add a new unit to it. Change the new unit's interface section to look like this:

```
type
  TAttemptedReopenEvent = reference to procedure (const CmdLine: string);

var
  OnAttemptedReopen: TAttemptedReopenEvent;

function CanRun: Boolean;
```

When assigned, `OnAttemptedReopen` will be called whenever a new instance of application is attempted to be run.

Next, move to the `implementation` section of the unit and add the following:

```
uses
  Winapi.Windows, Winapi.Messages, System.SysUtils,
  System.Classes, System.SyncObjs, Vcl.Forms;

const
  AppIdent = 'MyMutexDemoApp';
  CM_GETMAINFORMHANDLE = WM_USER + 1;

var
  HiddenWnd: HWND;
  Mutex: TMutex;

function CanRun: Boolean;
begin
  Result := (Mutex <> nil);
end;
```

Since we will be calling the Windows API directly, we need to use relevant `Winapi.*` units. Each 'window' at an API level has a 'procedure' (actually a function in Delphi code) that is called by the operating system on every 'message' — rather than having methods that can be invoked, a window responds to messages that are 'sent' or 'posted' to it. Here's what our custom window procedure looks like — place it immediately below the implementation of `CanRun`:

```
function HiddenWndProc(Handle: HWND; Msg: UINT;
  wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
var
  MainFormWnd: HWND;
  S: string;
  Struct: PCopyDataStruct;
begin
  case Msg of
    CM_GETMAINFORMHANDLE: Result := Application.MainFormHandle;
    WM_COPYDATA:
    begin
      //de-minimise the main form if necessary
      MainFormWnd := Application.MainFormHandle;
      if MainFormWnd <> 0 then
        if IsIconic(MainFormWnd) then OpenIcon(MainFormWnd);
      Result := 1;
      //invoke the callback if one is assigned
      if Assigned(OnAttemptedReopen) then
        try
          Struct := PCopyDataStruct(lParam);
          SetString(S, PChar(Struct.lpData),
            Struct.cbData div SizeOf(Char));
          OnAttemptedReopen(S);
        except
          Application.HandleException(ExceptObject);
        end;
    end
  else
    Result := DefWindowProc(Handle, Msg, wParam, lParam);
  end;
end;
```

CM_GEMAINFORMHANDLE is a custom message we have defined (anything with a numeric identifier over WM_USER is OK to be used for internal purposes). WM_COPYDATA, in contrast, is a special message Windows defines to allow passing small amounts of data between processes — normally, sending arbitrary data between processes won't work, since different processes have different memory spaces. Lastly, we pass on any other message received to the default window procedure.

The final code for the unit involves the mutex and creating (or sending messages to) the hidden window as necessary:

```
procedure DoInitialization;
var
  Data: TCopyDataStruct;
begin
  Mutex := TMutex.Create(nil, False, AppIdent);
  if Mutex.WaitFor(0) = wrSignaled then
  begin
    HiddenWnd := AllocateHWnd(nil);
    SetWindowText(HiddenWnd, AppIdent);
    SetWindowLongPtr(HiddenWnd, GWL_WNDPROC,
      LONG_PTR(@HiddenWndProc));
    Exit;
  end;
  FreeAndNil(Mutex);
  HiddenWnd := FindWindow(nil, AppIdent);
  if HiddenWnd = 0 then Exit;
  SetForegroundWindow(SendMessage(HiddenWnd,
    CM_GETMAINFORMHANDLE, 0, 0));
  Data.cbData := StrLen(CmdLine) * SizeOf(Char);
  Data.lpData := CmdLine;
  SendMessage(HiddenWnd, WM_COPYDATA, 0, LPARAM(@Data));
  HiddenWnd := 0;
end;

initialization
  DoInitialization;
finalization
  Mutex.Free;
  if HiddenWnd <> 0 then
  begin
    SetWindowLongPtr(HiddenWnd, GWL_WNDPROC,
      LONG_PTR(@DefWindowProc));
    DeallocateHWnd(HiddenWnd);
  end;
```

If the mutex can be acquired, the hidden window is created and the custom window procedure assigned to it (while we could call the RegisterWindowClass and CreateWindowEx API functions directly, using Delphi's AllocateHWnd function saves on a few lines). Alternatively, if the mutex couldn't be acquired, the hidden window of the original instance is sought, that instance activated, and the command line of the new instance passed to it.

If from the IDE you now repeatedly invoke the 'Run without Debugging' command (it's the toolbar button with the green arrow that doesn't have a red ladybird on it), you'll find nothing has actually been restricted! This is because nothing checks CanRun. So, close all running instances, open up the DPR in the IDE (Project|View Source), and add a suitable line at the top of the begin block:

```
begin
  if not CanRun then Exit; //!!!added
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  //etc.
```

Let's also update the caption of the form each time a new instance starts to run. So, head for the form's code, add the unit we've been working on to its uses clause, and handle the OnCreate event like this:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  OnAttemptedReopen :=
    procedure (const ACommandLine: string)
    begin
      Caption := 'Attempted new instance at ' + TimeToStr(Time));
    end;
end;
```

If you now try re-running the application, you'll find only one instance of it is indeed allowed. On subsequent attempts, the original instance is brought to the foreground, and de-minimised if necessary.

Read/write synchronisers (TMultiReadExclusiveWriteSynchronizer)

While the full name of `TMultiReadExclusiveWriteSynchronizer` (`TMREWSync` for short) is quite involved, its rationale is fairly simple: frequently, when the same resource is shared between multiple threads, there will be a lot more reading going on than writing. So long as this reading doesn't involve encapsulated writing (e.g., reading a property of an object could in principle involve fields being changed in the property getter), then religiously synchronising reads would cause threads to be blocked 'unnecessarily', i.e. during times when no thread is wishing to write as well.

`TMREWSync`, then, is a special sort of critical section class for just this sort of situation: at any one time, requests for a 'read lock' will be granted so long as no write lock is active. When a write lock is requested, in contrast, the thread blocks until all existing read (or write) locks are relinquished, at which point the write lock is granted and any request to read (or another thread's request to write) will be blocked.

While its implementation is a bit tricky, the public interface of `TMREWSync` is very simple. Just four methods are exposed:

```
procedure BeginRead;
procedure EndRead;
function BeginWrite: Boolean;
procedure EndWrite;
```

Unusually, `TMREWSync` also implements a matching interface, `IReadWriteSync`, that is constituted by the same methods. This is useful, since if an instance is shared only by secondary threads, then they can receive an `IReadWriteSync` reference in their constructors, store it in a field, and have interface reference counting take care of the synchroniser's destruction. Otherwise, you need to be careful the synchroniser doesn't get freed in (say) the main thread while it is still being used by another thread.

In use, you call `BeginRead` to acquire a read lock at the start of a `try`/`finally` block, do what you need to do to the shared resource in between the `try` and `finally`, before calling `EndRead` after the `finally`:

```
Synchronizer.BeginRead;
try
  //read protected resource...
finally
  Synchronizer.EndRead;
end;
```

Use of `BeginWrite` is similar, and can be called both while a read lock has already been granted, in which case the read lock is 'promoted' to a write lock, or when it hasn't. In the former case, nesting will be necessary so that each `BeginXXX` call is matched with a corresponding `EndXXX` one:

```
Synchronizer.BeginRead;
try
  //read protected resource...
  Synchronizer.BeginWrite;
  try
    //now write to it
  finally
    Synchronizer.EndWrite;
  end;
finally
  Synchronizer.EndRead;
end;
```

`BeginWrite` as much as `BeginRead` will block until the requested lock is actually granted. Despite that, it has a return value. This however indicates not success or failure, but whether another thread definitely did *not* get in and acquire a write lock while you were waiting for one yourself (a return value of `False`, then, indicates data previously retrieved from the shared resource might be out of date).

That it is possible for `False` to be returned is because any read lock is in fact relinquished while a thread waits to acquire a write lock (it is restored afterwards). This design helps prevent deadlocks when two threads currently holding a read lock request it be promoted to a write one.

TMREWSync on OS X

Unfortunately, `TMultiReadExclusiveWriteSynchronizer` when targeting OS X is implemented as a simple wrapper round `TMonitor`. While this means you can use the same code between Windows and OS X, the optimisations a genuine `TMREWSync` implementation would provide don't happen on the Mac.

Despite this, the POSIX API layer does in fact provide functionality almost identical to `TMREWSync` — so close, in fact, that writing a `TPosixMREWSync` class is very simple, especially given Delphi provides all the needed header translations in the box:

```
uses
  System.SysUtils, Posix.SysTypes, Posix.Pthread;

type
  TPosixMREWSync = class(TInterfacedObject, IReadWriteSync)
  strict private
    FHandle: pthread_rwlock_t;
  public
    constructor Create;
    destructor Destroy; override;
    procedure BeginRead;
    procedure EndRead;
    function BeginWrite: Boolean;
    procedure EndWrite;
  end;

constructor TPosixMREWSync.Create;
begin
  inherited Create;
  CheckOSError(pthread_rwlock_init(FHandle, nil));
end;

destructor TPosixMREWSync.Destroy;
begin
  if FHandle <> 0 then pthread_rwlock_destroy(FHandle);
  inherited;
end;

procedure TPosixMREWSync.BeginRead;
begin
  CheckOSError(pthread_rwlock_rdlock(FHandle));
end;

function TPosixMREWSync.BeginWrite: Boolean;
begin
  CheckOSError(pthread_rwlock_wrlock(FHandle));
  Result := False; //can't tell, so do the prudent thing
end;

procedure TPosixMREWSync.EndRead;
begin
  CheckOSError(pthread_rwlock_unlock(FHandle));
end;

procedure TPosixMREWSync.EndWrite;
begin
  CheckOSError(pthread_rwlock_unlock(FHandle));
end;
```

The only limitation compared to the 'real' `TMREWSync` is that the POSIX API does not indicate whether the promotion of a read to a write lock proceeded without another thread getting in first. Because of that, we have `BeginWrite` prudently always return `False`.

TMREWSync issues

While the idea behind read/write synchronisers can make them seem a very valuable synchronisation tool, their use in practice is limited by how they work with a very 'pure' meaning of 'read-only' access. When the resource in question is a variable, things are fine, however when it is an object property, all bets are off unless the object has been explicitly coded to work well in a multithreaded context.

This tension is fundamental, since one of the main purposes of using objects, and within that, properties rather than public fields, is to obscure both the origins of underlying data and the mechanics of retrieving it. While most properties directly map to private fields, this doesn't have to be the case:

- A property might 'lazy-load' its value in a fashion that causes related fields in the class or record to be written to the first time it is accessed. For example, the value might come from a file, the opening of which (and subsequent setting of corresponding fields) only happening the first time the property is read.
- As a variant of this, a property getter might cache its value, i.e. write the value to an internal field so that this field rather than the ultimate backing store (the file or whatever) needs accessing the next time the property is read.
- A property might be a specific case of a more general one, in which the getter first changes one or more associated properties, reads a value, then resets the properties it just changed. The `CommaText` property of `TStrings` is an example of this, its getter working by manipulating the `Delimiter`, `QuoteChar`, and `DelimitedText` properties of the object.

Insofar as you can guarantee a shared resource can be accessed in a read-only fashion 'all the way down', `TMultiReadExclusiveWriteSynchronizer` can be a useful tool in the toolbox. Otherwise, its slight performance hit compared to ordinary critical sections will mean it won't be as useful as it at first appears.

Spin waiting (TSpinWait)

When a thread is blocked, it is a very patient waiter — unlike a blocked person in a hurry, a blocked thread expends very little energy: it just sits there, idle. Nonetheless, there is a slight overhead involved in first entering the blocking state, then when the reason for waiting has ended, becoming unblocked.

Because of this, in situations of very high concurrency, it can be preferable not to block, but to 'spin'. This is the multithreading equivalent of running on the spot: doing nothing furiously.

```
while SomeFlag do { just loop };
```

When CPU resources are at a premium — the extreme case being when there is only one CPU core available — spinning doesn't make sense, since from the processor's point of view, an empty loop that just goes round and round is indistinguishable from code that finds a cure for cancer, deciphers Tony Blair's post-office financial affairs, or performs any other lengthy yet meaningful calculation. Consequently, code that just burns processor cycles will be allocated no more and no less processing resources than anything else. Nonetheless, if the time waiting for a certain condition is likely to be very short, and there are the cores available for the thread being waited on to do its business, 'spinning' rather than 'blocking' can be more performant.

For such as situation, the `System.SyncObjs` unit provides a couple of record types, `TSpinWait`, and `TSpinLock`. The first has the following public interface:

```
class procedure SpinUntil(const ACondition: TFunc<Boolean>);
class function SpinUntil(const ACondition: TFunc<Boolean>;
  Timeout: LongWord): Boolean;
class function SpinUntil(const ACondition: TFunc<Boolean>;
  const Timeout: TTimeSpan): Boolean;

procedure Reset;
procedure SpinCycle;
property Count: Integer read FCount;
property NextSpinCycleWillYield: Boolean read GetNextSpinCycleWillYield;
```

When using the instance methods, their pattern of use is generally as thus:

```
var
  Wait: TSpinWait;
begin
  Wait.Reset;
  while not <some condition> do Wait.SpinCycle;
  //do something...
```

`Reset` sets the internal spin counter to 0, which then gets incremented each time `SpinCycle` completes. Unless there is only one CPU core, the current value of the counter determines what `SpinCycle` actually does. Simplifying slightly, if it is 10 or less, a very short period of spinning is performed, otherwise the thread either 'yields' or 'sleeps' for a nominal amount of time. Both 'yielding' and 'sleeping' mean, in essence, letting another thread do its thing — where yielding only lets another thread on the same CPU core have a go, 'sleeping' does not have this restriction. To learn whether the next call to `SpinCycle` will yield or sleep, read the `NextSpinCycleWillYield` property.

The `SpinUntil` static method wraps the logic you would use when using the instance methods in one handy function. To it you pass a parameterless function that tests whether the condition you are interested in has come to pass, and (optionally) a timeout in milliseconds. While it is optional, you should generally always pass a timeout given you don't want to be spinning for long:

```
uses
```

```
  System.SysUtils, System.Classes, System.SyncObjs;

function CalcFibonacci(const Num: Integer): Int64;
begin
  if Num <= 1 then
    Result := Num
  else
    Result := CalcFibonacci(Num - 1) + CalcFibonacci(Num - 2);
end;

const
  Nth = 36; ExpectedResult = 14930352; Timeout = 200;
var
  ActualResult: Int64 = High(Int64);
  Thread: TThread;
  PassedTest: Boolean;
begin
  Thread := TThread.CreateAnonymousThread(
    procedure
    begin
      ActualResult := CalculateFibonacci(Nth);
    end);
  Thread.Start;
  PassedTest := TSpinWait.SpinUntil(
                  function : Boolean
                  begin
                    Result := (ActualResult = ExpectedResult)
                  end, Timeout);
  WriteLn('Passed test? ', PassedTest);
end.
```

The callback function will be repeatedly called in between spin cycles until it returns `True` (in which case `SpinUntil` itself will return `True`), or times out (in which case `SpinUntil` will return `False`).

Spin locks (TSpinLock)

`TSpinLock` uses `TSpinWait` to implement a spinning-based critical section-like type. It has the following public interface:

```
constructor Create(EnableThreadTracking: Boolean);

procedure Enter; inline;
function TryEnter: Boolean; overload; inline;
function TryEnter(Timeout: LongWord): Boolean; overload;
function TryEnter(const Timeout: TTimeSpan): Boolean; overload;

procedure Exit(PublishNow: Boolean = True);

property IsLocked: Boolean read GetIsLocked;
property IsLockedByCurrentThread: Boolean
  read GetIsLockedByCurrentThread;
property IsThreadTrackingEnabled: Boolean
  read GetIsThreadTrackingEnabled;
```

When thread tracking is enabled by passing `True` to the constructor, some extra checks are added so that invalid `Exit` calls will raise an exception and the `IsLockedByCurrentThread` property will work. When thread tracking is disabled, in contrast, invalid `Exit` calls will simply invalidate the structure in unpredictable ways, and `IsLockedByCurrentThread` will not return anything meaningful.

Whether thread tracking is enabled or not, spin locks are never 're-entrant'. This means it is invalid for a thread to nest `Enter` or `TryEnter` calls, something that is however perfectly legal if using a critical section or monitor. This limitation shouldn't really be an issue though, since nesting would indicate you expect to hold onto the outer lock for more than a very short period of time, in which case a spin lock wouldn't be an appropriate primitive to use in the first place.

Volatility (MemoryBarrier)

At the start of this section we met the concept of 'atomicity', in which an operation is said to be 'atomic' only if it will be performed without interruption. A related but distinct concept is that of 'volatility'. This concerns whether the result of a write operation will be immediately readable by another thread or not. That it may not is because the new result could in principle be stuck in some sort of low-level cache, or alternatively, have been subject to the CPU or compiler reordering instructions for optimisation purposes.

One way to ensure a write operation will be immediately readable by another thread is to call the `MemoryBarrier` function, as declared in the `System` unit. However, use of any synchronisation or signalling mechanism, or for that matter

`TInterlocked`, will do this implicitly. Furthermore, a combination of the CPU architectures Delphi compiles for (x86 and x64) and the way the Delphi compiler itself goes about its business makes the need to call `MemoryBarrier` essentially theoretical rather than practical. This may change when the Delphi compiler comes to target other architectures, but even then, only low level code that avoids use of normal locking techniques will be affected.

Lock-free synchronisation (TInterlocked.CompareExchange)

The final synchronisation primitive provided by the Delphi RTL is the `CompareExchange` method of `TInterlocked`. This is not, strictly speaking, a synchronisation mechanism by itself; rather, you use it to implement lock-free synchronisation with respect to changing a particular variable.

In all, `CompareExchange` is overloaded for the `Pointer`, `Integer`, `Int64`, `TObject`, `Double` and `Single` types; it also has a generic overload for classes. Its general form is the following:

```
class function CompareExchange(var Target: Type; NewValue: Type;
  Comparand: Type): Type; static;
```

Here, `Target` is the variable to be changed, `NewValue` the thing to change it too, and `Comparand` the expected current value of `NewValue`. As executed, `CompareExchange` performs the following atomically:

```
Result := Target;
if Target = Comparand then Target := NewValue;
```

One use for this is when implementing a function that returns a singleton object, i.e. an instance of a class that is supposed to have only one instance:

```
interface

type
  TMyObject = class
  //...
  end;

function Singleton: TMyObject;

implementation

var
  FSingleton: TMyObject;

function Singleton: TMyObject;
var
  NewInst: TMyObject;
begin
  if FSingleton = nil then
  begin
    NewInst := TMyObject.Create;
    if TInterlocked.CompareExchange<TMyObject>(FSingleton,
      NewInst, nil) <> nil then NewInst.Free;
  end;
  Result := FSingleton;
end;
```

The point of `CompareExchange` here is to avoid the `FSingleton` variable being assigned multiple times if and when the `Singleton` function comes to be called simultaneously by different threads — if `CompareExchange` doesn't return `nil`, then that means another thread has got in and initialised `FSingleton` in between our finding it `nil` and trying to assign it ourselves.

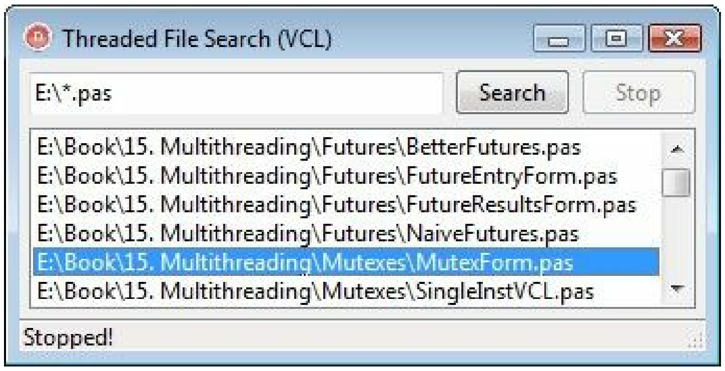
Inter-thread communication

Posting data to the main thread

When using multiple threads in a desktop application, a common need for interthread communication is to allow a background thread to notify the main thread of progress. One way this might be done is to use either `TThread.Synchronize` or `TThread.Queue`, passing an anonymous method that manipulates UI objects directly. This is rarely the best technique however, since it ties the thread code to specific visual controls, limiting the possibilities for code reuse and causing difficulties if and when the user interface comes to be redesigned.

An alternative approach is for the worker thread to fill a list object of some sort (typically a `TList` or `TQueue`) as and when it has data to pass back; the main thread then periodically checks the list or queue for new items, which it can do using a `TTimer` component. In both cases, access to the queue is protected by a critical section or monitor.

Amongst this book's sample code are two variants of a file finder program that demonstrates this technique, one using the VCL and the other FireMonkey:



E:\Book\15. Multithreading\Futures\BetterFutures.pas
E:\Book\15. Multithreading\Futures\FutureEntryForm.pas
E:\Book\15. Multithreading\Futures\FutureResultsForm.pas
E:\Book\15. Multithreading\Futures\NaiveFutures.pas
E:\Book\15. Multithreading\Mutexes\MutexForm.pas
E:\Book\15. Multithreading\Mutexes\SingleInstVCL.pas
Stopped!

The source for the background thread is shared by both projects. The substance of it calls the `Find`xxx API, with the paths of found files being passed back to the main thread in batches:

```
procedure TFileSearchThread.Execute;
begin
  FStopwatch := TStopwatch.StartNew;
  SearchDir(FSearchPath);
  if not Terminated then
    PostBatch;
end;

procedure TFileSearchThread.SearchDir(const Path: string);
var
  Rec: TSearchRec;
begin
  if Terminated then Exit;
  SetCurrentDirectory(ExcludeTrailingPathDelimiter(Path));
  //find files in the immediate directory...
  if FindFirst(Path + FSearchFileSpec,
     faAnyFile and not faDirectory, Rec) = 0 then
  try
    repeat
      FCurrentBatch.Add(Path + Rec.Name);
      if FStopwatch.ElapsedMilliseconds > FBatchPeriodInMSecs then
      begin
        PostBatch;
        FStopwatch.Reset;
        FStopwatch.Start;
      end;
    until Terminated or (FindNext(Rec) <> 0);
  finally
    FindClose(Rec);
  end;
  //find sub-directories and recurse
  if FindFirst(Path + '*.*', faDirectory, Rec) = 0 then
  try
    repeat
      if (Rec.Name <> '.') and (Rec.Name <> '..') then
        SearchDir(Path + Rec.Name + PathDelim);
    until Terminated or (FindNext(Rec) <> 0);
  finally
    FindClose(Rec);
  end;
```

```
end;
```

Batches are posted by calling a user-defined anonymous method. This leaves the mechanics of posting up to the calling code:

```
type
  TPostBatchProc = reference to procedure (ABatch: TEnumerable<string>);

  TFileSearchThread = class(TThread)
  strict private
    FBatchPeriodInMSecs: LongWord;
    FCurrentBatch: TList<string>;
    FPostBatchProc: TPostBatchProc;
  //...
    procedure PostBatch;
  public
    constructor Create(const ASearchSpec: string;
      ABatchPeriodInMSecs: LongWord;
      const APostBatchProc: TPostBatchProc);
  //...
  end;

procedure TFileSearchThread.PostBatch;
begin
  FPostBatchProc(FCurrentBatch);
  FCurrentBatch.Clear;
end;
```

In the VCL version, a `TListBox` is used is used for output. Batches sent by the background thread are put in a queue, which is then unloaded when the timer fires:

```
procedure TfrmFileSearchVCL.btnSearchClick(Sender: TObject);
begin
  btnSearch.Enabled := False;
  lsbFoundFiles.Items.Clear;
  StatusBar.SimpleText := 'Searching...';
  FWorkerThread := TFileSearchThread.Create(edtSearchFor.Text,
    tmrCheckForBatch.Interval,
    procedure (const ABatch: TEnumerable<string>)
    var
      Arr: TArray<string>;
    begin
      Arr := ABatch.ToArray;
      MonitorEnter(FFoundFilesQueue);
      try
        FFoundFilesQueue.Enqueue(Arr);
      finally
        MonitorExit(FFoundFilesQueue);
      end;
    end);
  FWorkerThread.OnTerminate := WorkerThreadTerminate;
  FWorkerThread.Start;
  tmrCheckForBatch.Enabled := True;
  btnStop.Enabled := True;
end;

procedure TfrmFileSearchVCL.tmrCheckForBatchTimer(Sender: TObject);
var
  Batch: TArray<string>;
  S: string;
begin
  if not MonitorTryEnter(FFoundFilesQueue) then Exit;
  try
    if FFoundFilesQueue.Count = 0 then Exit;
    Batch := FFoundFilesQueue.Dequeue;
  finally
    MonitorExit(FFoundFilesQueue);
  end;
  lsbFoundFiles.Items.BeginUpdate;
  try
    for S in Batch do
      lsbFoundFiles.Items.Add(S);
  finally
    lsbFoundFiles.Items.EndUpdate;
  end;
end;
```

In the FireMonkey version, a FMX `TGrid` control is used for the display. Since this control doesn't carry its own data (instead, you handle the `OnGetValue` event to tell it what text to show), a separate `TList<string>` is needed to hold all the matched file names. In principle, the callback method used for posting batches could add to this main list directly. However, using a separate list as an intermediary means we keep a lid on the number of places we must remember to lock:

```
procedure TfrmFileSearchFMX.btnSearchClick(Sender: TObject);
begin
  btnSearch.Enabled := False;
  grdFoundFiles.RowCount := 0;
  FFoundFiles.Clear;
  lblStatus.Text := 'Searching...';
  FWorkerThread := TFileSearchThread.Create(edtSearchFor.Text,
    tmrCheckForBatch.Interval,
    procedure (const ABatch: TEnumerable<string>)
    begin
      TMonitor.Enter(FNextBatch);
      try
        FNextBatch.AddRange(ABatch);
      finally
        TMonitor.Exit(FNextBatch);
      end;
    end);
  FWorkerThread.OnTerminate := WorkerThreadTerminate;
  FWorkerThread.Start;
  tmrCheckForBatch.Enabled := True;
  btnStop.Enabled := True;
end;

procedure TfrmFileSearchFMX.tmrCheckForBatchTimer(Sender: TObject);
begin
  if not TMonitor.TryEnter(FNextBatch) then Exit;
  try
    FFoundFiles.AddRange(FNextBatch);
    FNextBatch.Clear;
  finally
    TMonitor.Exit(FNextBatch);
  end;
  grdFoundFiles.RowCount := FFoundFiles.Count;
end;
```

Producer/consumer queues (TThreadedQueue)

An alternative to using an ordinary `TQueue` in combination with `TMonitor` or a critical section object is `TThreadedQueue`. Declared in `System.Generics.Collections`, this implements a 'producer/consumer' queue. Aside from protecting additions and removals with a lock, it also sets a fixed cap on the number of items it can hold, making a thread trying to add an item wait if the capacity has been reached, and conversely, making a thread trying to remove an item wait too if there is nothing to remove. In a normal `TQueue`, in contrast, the 'capacity' determines only the number of slots currently allocated, not the actual maximum number of items; furthermore, attempting to dequeue an item from an empty `TQueue` is considered an error, not a request that should be eventually fulfilled.

In use, you specify the maximum number of slots, together with millisecond timeouts for both pushing and popping items, when creating the threaded queue. By default, 10 slots are allocated and neither pushing nor popping may timeout:

```
constructor Create(AQueueDepth: Integer = 10;
  PushTimeout: LongWord = INFINITE;
  PopTimeout: LongWord = INFINITE);
```

If either pushing or popping should never cause the calling thread to block, pass 0 for the timeout:

```
var
  StringQueue: TThreadedQueue<string>;
begin
  StringQueue := TThreadedQueue<string>.Create(
    20, 0, 0); //20 slots, no blocking
```

Where `TQueue` has `Enqueue` and `Dequeue` methods, `TThreadedQueue` has `PushItem` and `PopItem`. Given the possibility of timeouts, their signatures are also a bit different, as well as being overloaded:

```
function PopItem: T; overload;
function PopItem(var AQueueSize: Integer): T; overload;
function PopItem(var AQueueSize: Integer;
  var AItem: T): TWaitResult; overload;
```

```
function PopItem(var AItem: T): TWaitResult; overload;

function PushItem(const AItem: T): TWaitResult; overload;
function PushItem(const AItem: T;
  var AQueueSize: Integer): TWaitResult; overload;
```

Usually the ones to use are the third variant of `PopItem` and the first variant of `PushItem`, since the current queue size shouldn't really bother the calling thread — just knowing whether the request was successful should be enough.

With respect to the result type, `TWaitResult` is an enumeration defined in `System.SyncUtils` with the possible values `wrSignaled`, `wrTimeout`, `wrAbandoned`, `wrError` and `wrIOCompletion`. However, only `wrTimeout` (meaning failure) and `wrSignaled` (meaning success) will be returned by `PushItem` or `PopItem`.

The remaining public members of `TThreadedQueue` are a method, `DoShutdown`, together with the read-only properties `QueueSize`, `ShutDown` (which reports whether `DoShutdown` has been called), `TotalItemsPopped` and `TotalItemsPushed`. Calling `DoShutdown` when one or more threads are currently blocking on either `PushItem` or `PopItem` will cause them to be released; usually you do this from the main thread when the application terminates, to give blocking threads a chance to terminate gracefully rather than just be abruptly terminated by the operating system:

```
procedure TWorkerThread.Execute;
var
  Work: string;
begin
  while WorkQueue.PopItem(Work) = wrSignaled do
  begin
    //process the work item...
  end;
  //clean up...
end

//...

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  WorkQueue.DoShutdown;
  WaitForWorkerThreads;
  WorkQueue.Free;
end;
```

TThread.Terminate, TThread.WaitFor

Any secondary thread you create in Delphi will be a 'background' thread. This means no secondary thread will keep the application running if the main thread has finished.

Because of that, the following program won't output anything:

```
program AbruptFinish;

{$APPTYPE CONSOLE}

uses System.Classes;

begin
  TThread.CreateAnonymousThread(
    procedure
    begin
      Sleep(1000);
      WriteLn('This will never execute!');
    end).Start;
end.
```

This behaviour is not intrinsically bad, since the operating system will reclaim any memory and resources used by the secondary threads it halts. Nonetheless, the particular requirements of the application may make it undesirable even so. For example, a thread might have data that should be flushed to disk, or a database transaction that needs completing.

If you need clean up code to run, putting it inside the thread object's destructor or even the `finally` part of a `try`/`finally` statement won't help, since the operating system will just ignore them — once the main thread has finished, secondary threads are halted there and then. Instead, the main thread must ask secondary threads to finish, then explicitly wait on them to respond or complete.

This pattern is implemented by the `TThread` class in the form of two methods, `Terminate` and `WaitFor`, and a property, `Terminated`: when the main thread wants a worker to abort, it calls the thread object's `Terminate` method. This sets the worker's `Terminated` property to `True`. In order for the thread to actually terminate, the `Execute` implementation must then periodically check the `Terminated` property, exiting once this returns `True` (in the case of an anonymous thread, the thread procedure must periodically check `TThread.CurrentThread.CheckTerminated`). When the thread is structured around either a `while` or `repeat`/`until` loop, this checking can be done easily enough:

```
//explicit TThread descendant + while loop
procedure TMyThread.Execute;
begin
  while not Terminated do
  begin
    //do stuff
  end;
end;

//anonymous TThread + repeat/until loop
Thread := TThread.CreateAnonymousThread(
  procedure
  begin
    repeat
      { do stuff }
    until TThread.CurrentThread.CheckTerminated;
  end);
```

Otherwise, the thread object's `Execute` method or the anonymous thread's anonymous procedure should make periodic checks:

```
procedure TMyThread.Execute;
begin
  //... do stuff
  if Terminated then Exit;
  //... do more stuff
  if Terminated then Exit;
  //... do yet more stuff
end;
```

Since `Terminate` just sets a flag rather than terminating the thread there and then, there needs to be a way to wait on a thread to finish. This is provided for by the `WaitFor` method:

```
var
  Worker: TThread;
begin
  //...
  Worker.Terminate;
  Worker.WaitFor;
```

There are a few problems with `WaitFor` however. The first is that you cannot specify a timeout, which means if the waited on thread hangs (gets in an endless loop, has deadlocked, etc.), then the calling thread will hang too. Secondly, it requires the thread's `FreeOnTerminate` property to be `False` — in set it to `True` (or in the case of an anonymous thread, leave it set to `True`), and you will get an odd-looking exception. Lastly, it does not work in a console program when the caller is the main thread. This is because `WaitFor` performs some special handling for the `Synchronize` method, which itself is incompatible with console programs.

To an extent, these problems can be overcome by calling the operating system API directly. In the case of Windows, this means calling `WaitForSingleObject`, a function declared in `Winapi.Windows`. Unlike `TThread.WaitFor`, this function always allows you to specify a millisecond timeout (to mimic the `WaitFor` behaviour, pass the special constant `INFINITE`):

```
uses Winapi.Windows, System.SysUtils, System.Classes;

var
  Worker: TThread;
begin
  //...
  Worker.Terminate;
  WaitForSingleObject(Worker.Handle, INFINITE);
```

If you have more than one secondary thread to wait on, you can wait on them all at once by calling `WaitForMultipleObjects`:

```
procedure TerminateAndWaitForThreads(
  const Threads: array of TThread; Timeout: LongWord = INFINITE);
var
  Handles: array of THandle;
  I: Integer;
begin
  SetLength(Handles, Length(Threads));
  for I := 0 to High(Handles) do
  begin
    Handles[I] := Threads[I].Handle;
    Threads[I].Terminate;
  end;
  WaitForMultipleObjects(Length(Handles), @Handles[0], True, Timeout);
end;
```

On OS X, the relevant API function is `pthread_join`. This is passed the thread ID rather than the thread handle (thread handles in fact don't exist on POSIX), and does not accept a timeout. There is also no equivalent to `WaitForMultipleObjects` — instead, you need to call `pthread_join` in a loop:

```
uses Posix.Pthread, System.SysUtils, System.Classes;

var
  I: Integer;
  Thread: TThread;
  Workers: array of TThread;
  ExitCode: Integer;
begin
  //...
  for Thread in Workers do
  begin
    Thread.Terminate;
    pthread_join(Thread.ThreadID, @ExitCode);
  end;
```

Even when calling the native API directly however, the problem of `FreeOnTerminate` when that property is set to `True` still largely remains, since the thread object may have been freed by the time the `Handle` or `ThreadID` property is read. In principle, the only safe option is to create the thread objects suspended, read off their handles or IDs into a variable, get them going, then when the time times, call `WaitForXXXObject` or `pthread_join` with the saved handles or IDs:

```
uses Winapi.Windows;

var
  WorkerThread: TThread;
  WorkerHandle: THandle;
begin
  WorkerThread := TThread.CreateAnonymousThread(
    procedure
    begin
      //...
    end);
  WorkerHandle := WorkerThread.Handle;
  WorkerThread.Start;
  //...
  WaitForSingleObject(WorkerHandle, 5000);
end.
```

Alternatively, you can avoid looking for a `TThread.WaitFor` substitute in the first place and use a separate signalling object instead. We'll look at perhaps the most apposite candidate next: `TCountdownEvent`.

Countdown events (TCountdownEvent)

`TCountdownEvent` (declared in `System.SyncObjs`) implements a thread-safe counter that starts with a specified value, can be incremented or decremented as necessary after creation, and has a `WaitFor` method which allows a thread to block until the counter falls to zero — when this happens, the countdown event is said to be 'set' or 'signalled'. Unlike `TThread.WaitFor`, `TCountdownEvent.WaitFor` allows specifying a timeout, works fine with `TThread.FreeOnTerminate`, and is safe to use in a console application.

The public interface of `TCountdownEvent` looks like this:

```
constructor Create(Count: Integer);
//methods
function Signal(Count: Integer = 1): Boolean;
procedure AddCount(Count: Integer = 1);
procedure Reset; overload;
procedure Reset(Count: Integer); overload;
function TryAddCount(Count: Integer = 1): Boolean;
function WaitFor(Timeout: LongWord = INFINITE): TWaitResult;
//properties
property CurrentCount: Integer read FCurrentCount;
property InitialCount: Integer read FInitialCount;
property IsSet: Boolean read GetIsSet;
```

The value passed the constructor must be zero or over; if zero, the object is signalled immediately, and the only way to unsignal it will be to call Reset with a value greater than zero. (When Reset is called without an argument, the initial count is used.)

Call either AddCount or TryAddCount to increment the counter; when the count is zero, the latter will return False and the former raise an exception. Calling Signal decrements the counter; attempt this when the object is already signalling (i.e., when the counter is zero) and an exception will be raised, and similarly, you cannot decrement by more than CurrentCount. In all other cases, Signal will return True when the counter has fallen to zero due to the requested decrement, and False otherwise.

In use, the main thing to get right about TCountdownEvent is where to call AddCount or TryAddCount and where to call Signal. This is because you don't want to let otherwise unconnected bugs in the threads that use the countdown event (e.g., an exception being raised in the middle of an Execute method) causing the object to never signal when another thread is waiting on it to do so.

When being used as a TThread.WaitFor replacement, one way is to call AddCount at the top of the thread object's Execute method (or thread procedure in the case of an anonymous thread), and ensure Signal is called by virtue of a try/finally block:

```
procedure TMyThread.Execute;
begin
  FCountdownEvent.AddCount;
  try
    //actual work here...
  finally
    FCountdownEvent.Signal;
  end;
end;
```

The countdown event object can be passed to the constructor, or in the case of an anonymous thread, captured by the anonymous method in the normal way.

A caveat with this approach is that it may cause the thread to signal 'too early' if the code that uses it has handled the thread object's OnTerminate event property. This is because any such handler will only be called *after* Execute completes, and therefore, after the thread has signalled it has finished.

If this is an issue, you should use an explicit TThread descendant, set FreeOnTerminate to True, and increment and decrement the countdown event in the constructor and destructor respectively:

```
type
  TWorker = class(TThread)
  strict private
    FCountdownEvent: TCountdownEvent;
  protected
    procedure Execute; override;
  public
    constructor Create(ACountdownEvent: TCountdownEvent);
    destructor Destroy; override;
  end;

constructor TWorker.Create(ACountdownEvent: TCountdownEvent);
begin
  inherited Create(True);
  ACountdownEvent.AddCount;
  FCountdownEvent := ACountdownEvent;
  FreeOnTerminate := True;
end;

destructor TWorker.Destroy;
begin
  if FCountdownEvent <> nil then FCountdownEvent.Signal;
  inherited;
end;
```

This will work because the order of execution is guaranteed to be Execute, OnTerminate handler, Destroy. If FreeOnTerminate is not set to True, you may get into the problem of wanting to call Free from the thread that is waiting on the countdown event. When FreeOnTerminate is True, however, the destructor will be called in the context of the secondary thread, just before it dies, which is what we want.

The last part of the puzzle is the code for the thread that waits on the workers:

```
const
  WorkerCount = 4;
var
```

```
  CountdownEvent: TCountdownEvent;
  I: Integer;
  S: string;
begin
  CountdownEvent := TCountdownEvent.Create(1);
  try
    for I := 1 to WorkerCount do
      TWorkerThread.Create(CountdownEvent).Start;
    //... do some other stuff, perhaps
    CountdownEvent.Signal;
    CountdownEvent.WaitFor;
  finally
    CountdownEvent.Free;
  end;
end.
```

Here, we initialise the countdown event's counter to 1, have each worker thread increment it on their construction, then decrement it immediately before waiting on the worker threads to do the same.

Semaphores (TSemaphore, TLightweightSemaphore)

Functionally, a 'semaphore' is a bit like a countdown event standing on its head: a thread-safe counter that blocks not *until* it falls to zero, but *when* it does. This functionality is useful in, for example, a multithreaded database application, in which the number of active database connections should be limited: if allocation of open connections is managed via a semaphore, then a worker thread can be made to block when the maximum of open connections is reached.

On creating a semaphore, you specify the initial count, which denotes the maximum number of slots that can be taken, and possibly a maximum count too. While the internal counter remains above zero, requests to 'acquire' the semaphore (take a slot, decrement the counter) are granted; once all slots are taken, subsequent requests then block until another thread 'releases' the semaphore (relinquishes a lock, and therefore increments the counter):

```
var
  I: Integer;
  NumWorkersLeft: Integer;
  ThrottleSemaphore: TLightweightSemaphore;
  WorkersSemaphore: TSemaphore;
begin
  WorkersSemaphore := nil;
  //max 3 slots
  ThrottleSemaphore := TLightweightSemaphore.Create(3);
  try
    //max 1 slot
    WorkersSemaphore := TSemaphore.Create;
    //grab the slot
    WorkersSemaphore.Acquire;
    //spin up the worker threads
    NumWorkersLeft := 10;
    for I := 1 to NumWorkersLeft do
      TThread.CreateAnonymousThread(
        procedure
        begin
          try
            ThrottleSemaphore.Acquire;
            try
              { do some work... }
            finally
              ThrottleSemaphore.Release;
            end;
          finally
            if TInterlocked.Decrement(NumWorkersLeft) = 0 then
              WorkersSemaphore.Release;
          end;
        end).Start;
    //wait on worker threads to finish
    WorkersSemaphore.Acquire;
  finally
    WorkersSemaphore.Free;
    ThrottleSemaphore.Free;
  end;
end.
```

In this example, two semaphores are used, one to throttle the worker threads and the other as a `TThread.WaitFor` substitute. In the first case, only three worker threads are allowed to do their stuff at any one time. In the second, the semaphore is created with only one slot available, which is immediately taken. This means that when `Acquire` is called after setting up the worker threads, the main thread will block. When the last worker thread finishes, it then releases the second semaphore to unblock the main thread.

Semaphores vs. critical section objects

On the fact of it, a semaphore can seem like a sort of critical section object that allows more than one lock to be taken at a time. Appearances can prove deceptive however, since semaphore acquisitions are not associated with or 'owned by' the threads that make them. One implication of this is that when a thread acquires a semaphore, it can be another thread that makes the corresponding release. More problematically, a further implication is that you should usually avoid nested acquisitions, i.e. when the same thread acquires a semaphore multiple times without releasing it in between, at least without understanding what this will mean.

For example, say a semaphore with a count of 1 is created. A thread then comes along and acquires it, does something, then requests to acquire it again:

```
var
  Semaphore: TLightweightSemaphore;
begin
  //Create the semaphore with one available slot
  Semaphore := TLightweightSemaphore.Create(1);
  try
    //Take that slot
    Semaphore.Acquire;
    try
      WriteLn('Taken first lock...');
      //Attempt to take it again...
      Semaphore.Acquire;
      try
        WriteLn('This will never be reached!');
      finally
        Semaphore.Release;
      end;
    finally
      Semaphore.Release;
    end;
  finally
    Semaphore.Free;
  end;
end.
```

If you run this code, you'll find the thread has deadlocked itself! Replace the semaphore with a critical section object, however, and the second 'acquisition' runs through fine:

```
var
  CriticalSection: TCriticalSection; //could use TMonitor instead
begin
  //A critical section has only one available slot by its very nature
  CriticalSection := TCriticalSection.Create;
  try
    //Take that slot
    CriticalSection.Acquire;
    try
      WriteLn('Taken first lock...');
      //Since locks are thread specific, calling Acquire
      //multiple times from the same thread is OK
      CriticalSection.Acquire;
      try
        WriteLn('We''ve made it!');
      finally
        CriticalSection.Release;
      end;
    finally
      CriticalSection.Release;
    end;
  finally
    CriticalSection.Free;
  end;
end.
```

This doesn't show semaphores are bad, just that they have their own distinctive semantics which need respecting.

Nonetheless, another sort of situation where semaphores not being 'owned' at all can bite is when the same semaphore is shared between multiple processes (we will be looking at this feature shortly). In being open to that possibility, semaphores on Windows are similar to mutexes. However, if a process takes ownership of a shared mutex and promptly crashes, the operating system will automatically release the lock on the crashed application's behalf. If a process takes ownership of a shared semaphore and terminates unexpectedly however, the lock is never released. Caveat emptor!

Normal vs. 'lightweight' semaphores

In `System.SyncObjs` there is not one but two semaphore classes, `TSemaphore` and `TLightweightSemaphore`. Where the first wraps a semaphore primitive provided by the underlying operating system API, the second is a custom implementation designed for situations when the semaphore is rarely blocked (i.e., when the internal counter is usually above zero). If that is the case, `TLightweightSemaphore` could be anywhere up to twice as performant as `TSemaphore`; however, if the semaphore is often blocked, this advantage will quickly fade and even reverse slightly.

As both classes descend from the abstract `TSynchroObject` class, and `TSynchroObject` introduces the `Acquire`, `Release` and `WaitFor` methods (`WaitFor` being the same as `Acquire`, only with a timeout specified), you can fairly easily switch between the two semaphore classes if and when desired. The main differences in their respective interfaces lie with their constructors. `TLightweightSemaphore` has a single constructor that looks like this:

```
constructor Create(AInitialCount: Integer;
  AMaxCount: Integer = MaxInt);
```

If `MaxCount` is specified and you attempt to increment the semaphore with multiple `Release` calls beyond what it has been set to, an exception is raised.

In contrast, `TSemaphore` has three constructors:

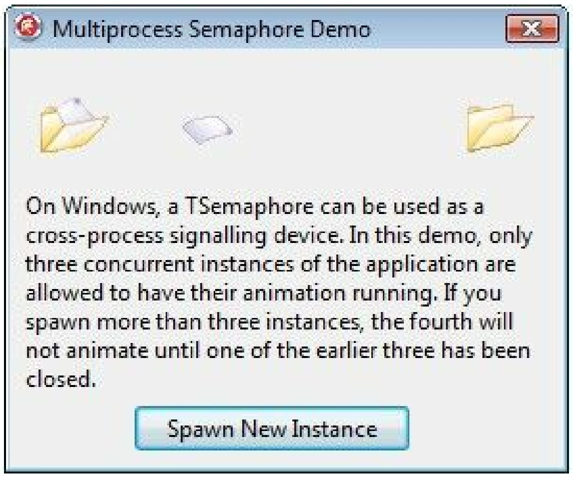
```
constructor Create(UseCOMWait: Boolean = False); overload;
constructor Create(SemaphoreAttributes: PSecurityAttributes;
  AInitialCount, AMaximumCount: Integer; const Name: string;
  UseCOMWait: Boolean = False); overload;
constructor Create(DesiredAccess: LongWord;
  InheritHandle: Boolean; const Name: string;
  UseCOMWait: Boolean = False); overload;
```

The first version just calls the second, passing `nil`, 1, 1, an empty string and `UseComWait`. Most of the additional parameters are Windows specific — if cross compiling for OS X, `Name` should always be an empty string, and `SemaphoreAttributes` and `UseCOMWait` are always ignored.

Interprocess semaphores (Windows only)

On Windows, semaphores created using `TSemaphore` can work across different processes. If you call the version of `Create` that takes an `InheritHandle` parameter, a semaphore with the name specified must have already been created. In contrast, the version that takes the initial and maximum counts will create the underlying operating system object if it doesn't already exist, otherwise it will be 'opened' and the `InitialCount` and `MaxCount` arguments ignored.

In this book's accompanying source code, you can find an example project (`MultiProcessSemaphore.dpr`) that demonstrates the second case. The project takes the form of a simple VCL application, the main form having a `TAnimate` control that only actually animates for a maximum of three concurrent instances of the application:



On Windows, a TSemaphore can be used as a cross-process signalling device. In this demo, only three concurrent instances of the application are allowed to have their animation running. If you spawn more than three instances, the fourth will not animate until one of the earlier three has been closed.

As implemented, this works through a worker thread being created on start up, which immediately attempts to acquire the shared semaphore; if it can, then the animation is activated, otherwise the thread blocks until either it the semaphore is acquired or the application has terminated. The form's `OnCreate` and `OnDestroy` handlers look like this:

```
procedure TfrmSemaphore.FormCreate(Sender: TObject);
begin
```

```
  { Set up an 'auto-release' event for terminating the worker
    thread gracefully (events will be discussed shortly) }
  FTerminateEvent := TEvent.Create(nil, False, False, '');
  FSemaphoreThread := TThread.CreateAnonymousThread(
    procedure
    var
      Objs: THandleObjectArray;
      Semaphore: TSemaphore;
      SignalledObj: THandleObject;
    begin
      Semaphore := TSemaphore.Create(nil, 3, 3, //Name semaphore
        ExtractFileName(ParamStr(0)));          //after EXE
      try
        Objs := THandleObjectArray.Create(Semaphore,
          FTerminateEvent);
        if (THandleObject.WaitForMultiple(Objs, INFINITE, False,
            SignalledObj) <> wrSignaled) or
           (SignalledObj <> Semaphore) then Exit;
        try
          TThread.Queue(TThread.CurrentThread,
            procedure
            begin
              AnimatedCtrl.Active := True;
            end);
          FTerminateEvent.WaitFor(INFINITE);
        finally
          Semaphore.Release;
        end;
      finally
        Semaphore.Free;
      end;
    end);
  FSemaphoreThread.FreeOnTerminate := False;
  FSemaphoreThread.Start;
end;

procedure TfrmSemaphore.FormDestroy(Sender: TObject);
begin
  FTerminateEvent.SetEvent; //Signal semaphore thread to quit
  FSemaphoreThread.WaitFor; //Wait on it quiting
  FSemaphoreThread.Free;    //Free it!
end;
```

A separate worker thread is used, because otherwise, the main thread will have to block when more than three instances are open, rendering the application unresponsive. This secondary thread then stays alive for the duration of the application so that it can release and free the semaphore itself. If it didn't, then the semaphore object would be used by both the worker and the main thread, and as a general rule, it is better to keep objects private to particular threads if you can.

Since the context is a GUI application, we can use `TThread.WaitFor` so long as we turn off `FreeOnTerminate`, which we do. However, another signalling primitive needs to be used — an 'event' — so that the worker thread can be woken *either* by the semaphore *or* the main thread. Let's turn to consider events now.

Events (TEvent and TLightweightEvent)

It doesn't happen often, but 'event' is one word that has two distinct meanings in Delphi programming. On the one hand, it refers to the event properties of an object (typically though not necessarily a component), e.g. the `OnClick` 'event' of a button control or the `OnTerminate` 'event' of `TThread;` on the other though, it refers to a multithreading signalling device.

In the second case, one or more threads 'wait' on an event to be signalled, which is done by another thread simply 'setting' the event. Once set, at least one of the waiters is let go. Whether the rest are too depends on whether the event was created to 'manually reset' or 'automatically reset'. With a 'manual reset' event, all waiting threads are unblocked until another thread explicitly 'resets' the event (i.e., closes it off again). In contrast, when an 'auto-reset' event is set, only one waiting thread is unblocked before the event closes itself off again.

Similar to semaphores, the Delphi RTL provides not one but two event implementations, `TEvent` (a wrapper for an operating system-provided primitive) and `TLightweightEvent` (a custom implementation partly based on spin locks). As with semaphores, the OS wrapper can be nameable and cross-process on Windows, while the 'lightweight' version should be more performant in cases when the object is mostly signalled (i.e., set). That said, `TLightweightEvent` only supports manual resets, and on Windows, a thread can wait on more than one `TEvent` at a time where it can't wait on more than one `TLightweightEvent` simultaneously.

Both `TEvent` and `TLightweightEvent` descend from `TSynchroObject`, and so share `Acquire` and `WaitFor` methods (the former calling the latter with a timeout of `INFINITE`). Beyond that, `TLightweightEvent`'s public interface is as thus:

```
//constructors
constructor Create;
constructor Create(InitialState: Boolean);
constructor Create(InitialState: Boolean; SpinCount: Integer);
//methods
procedure ResetEvent;
procedure SetEvent;
//properties
property BlockedCount: Integer read FBlockedCount;
property IsSet: Boolean read GetIsSet;
property SpinCount: Integer read GetSpinCount;
```

The first version of `Create` simply calls the second, passing `False` for `InitialState`. This means the event is initialised unset (i.e., blocking). Generally, the default for `SpinCount` (1 on a single core machine, 10 on a multicore one) is fine, or at least, not work fiddling with, but you can tweak if you want by calling the third version of `Create`. Of the other methods, `SetEvent` unblocks waiters and `ResetEvent` blocks them again.

The interface of `TEvent` is similar, only without the properties:

```
constructor Create(UseCOMWait: Boolean = False); overload;
constructor Create(EventAttributes: PSecurityAttributes;
  ManualReset, InitialState: Boolean; const Name: string;
  UseCOMWait: Boolean = False); overload;
procedure SetEvent;
procedure ResetEvent;
```

The first version of the constructor calls the second, passing `nil`, `True`, `False` and an empty string for the first three parameters. As with mutexes and semaphores, named events are for cross process signalling, and only supported on Windows.

Of the various synchronisation primitives available, events are one of the easier to grasp and use:

```
uses
  System.SysUtils, System.Classes, System.SyncObjs;

var
  Event: TEvent;
begin
  //create an auto-reset event that is initially unset
  Event := TEvent.Create(nil, False, False, '');
  try
    TThread.CreateAnonymousThread(
      procedure
      begin
        WriteLn('Background thread will now wait on event...');
        if Event.WaitFor(5000) = wrSignaled then //5 secs timeout
        begin
          WriteLn('Well that took a long time!');
          Event.SetEvent;
        end
        else
          WriteLn('Urgh, something went wrong...');
      end).Start;
    Sleep(4000);
    Event.SetEvent; //let the waiting thread go
    Event.WaitFor;  //now wait on the event ourselves
  finally
    Event.Free;
  end;
  Write('All done now');
end.
```

In this example, the event object is first created as an auto-reset event that is unset (non-signalling) at the off. Next, a background thread is created and run, which waits for up to five seconds for the event to signal. When it does, a message is outputted, and the event set once more. Back in the main thread, we dawdle for four seconds after creating the secondary thread, before setting the event to release the other thread and immediately waiting on the event ourselves. Once the background thread has received and reset the event, the main thread is woken, outputs a completed message, and exits.

Waiting on multiple 'handle' objects (Windows only)

On Windows, all of `TEvent`, `TMutex` and `TSemaphore` have ‘handles’, allowing a thread to wait on more than one of them at the same time. At the Windows API level, this is done by calling either `WaitForMultipleObjects`, `WaitForMultipleObjectsEx` or the COM equivalent, `CoWaitForMultipleHandles`, as appropriate. While any of these routines could be called directly, the `THandleObject` class provides a handy `WaitForMultiple` class method that wraps them nicely:

```
type
  THandleObjectArray = array of THandleObject;
  TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError,
    wrIOCompletion);

class function WaitForMultiple(const Objs: THandleObjectArray;
  Timeout: LongWord; All: Boolean; out Signaled: THandleObject;
  UseCOMWait: Boolean = False; Len: Integer = 0): TWaitResult;
```

The `Timeout` parameter specifies how many milliseconds the function will wait for the objects to signal; as with the regular `WaitFor` methods, pass `INFINITE` if you do not want to ever time out. When `All` is `True`, the function only returns when *every* object listed has signalled or aborted, otherwise the function returns when at least one of the objects has. On return, `Signaled` will contain the first object that signalled. With respect to the optional parameters, `UseCOMWait` determines whether `CoWaitForMultipleHandles` rather than `WaitForMultipleObjectsEx` is used under the hood, and when set, `Len` determines the number of objects in the source array to wait for (if `Len` is left as 0, all of them will be).

A common (though by no means only) use of `WaitForMultiple` is in situations where a thread wishes to wait on both the thing it is interested in *and* a separately-managed cancellation event. This was the case in the interprocess semaphore demo presented earlier:

```
var
  Objs: THandleObjectArray;
  Semaphore: TSemaphore;
  SignalledObj: THandleObject;
begin
  //...
  Objs := THandleObjectArray.Create(Semaphore, FTerminateEvent);
  if (THandleObject.WaitForMultiple(Objs, INFINITE, False,
      SignalledObj) <> wrSignaled) or
     (SignalledObj <> Semaphore) then Exit;
```

With `AAll` set to `False`, `WaitForMultiple` returns as soon as one of the objects waited on signals; if that was the termination event, then the thread gracefully exits.

Condition variables (TConditionVariableMutex, TConditionVariableCS)

Being based on Unix, OS X implements the POSIX threading API (‘pthreads’). In itself, this API supports neither Windows-style threading events nor the ability to wait on more than one signalling object at the same time. Instead, it has ‘condition variables’. These work in harness with a locking object of some sort: in use, a thread first acquires the lock, before waiting on the condition variable to signal or ‘pulse’. Internally, going into the waiting state involves the lock being released, allowing other threads — potentially other waiters, but ultimately the thread that will be doing the pulsing — to take the lock themselves. Once pulsed, a thread is woken and given back the lock.

The role of the condition variable in all this is to enforce atomicity, both with respect to releasing the lock to go into a waiting state, and being woken with the lock reacquired. The condition variable also ensures waiting threads are woken in an orderly fashion, if more than one is pulsed at the same time.

As POSIX doesn’t support Windows-style events, so the Windows API did not originally support condition variables. Latterly Microsoft has added a condition variable primitive however. Nonetheless, there is still a slight difference, since where the locking object used with a pthreads condition variable is a mutex, the one used with the Windows API is a critical section. Consequently, `System.SyncObjs` provides *two* condition variable classes, `TConditionVariableMutex` for working with a `TMutex` and `TConditionVariableCS` for working with a `TCriticalSection` or `TRTLCriticalSection`. By backfilling implementations as necessary, these two classes work on all supported platforms, including older versions of Windows that don’t natively support condition variables at all.

Here’s a simple example of `TConditionVariableMutex` in use:

```
uses
  System.SysUtils, System.Classes, System.SyncObjs;

var
  Lock: TMutex;
  CondVar: TConditionVariableMutex;
  I: Integer;
begin
```

```
  Lock := TMutex.Create;
  CondVar := TConditionVariableMutex.Create;
  try
    for I := 1 to 2 do
      TThread.CreateAnonymousThread(
        procedure
        begin
          Lock.Acquire;
          try
            CondVar.WaitFor(Lock);
            WriteLn('Thread ' +
              IntToStr(TThread.CurrentThread.ThreadID) +
                ' got signalled and will now keep the ' +
                ' look for 2s...');
            Sleep(2000);
            WriteLn('... about to release the lock');
          finally
            Lock.Release;
          end;
        end).Start;
      WriteLn('Condition variable will be signalled in 1s...');
      Sleep(1000);
      CondVar.ReleaseAll; //signal to all waiters
      ReadLn;
  finally
    CondVar.Free;
    Lock.Free;
  end;
end.
```

In this example, two background threads are started up. Both seek to acquire the lock before immediately waiting on the condition variable — in practice, the first thread calling `WaitFor` will allow the second to get past the `Acquire` stage and call `WaitFor` itself. After dawdling for a short period of time, the main thread signals to all waiting threads, causing the two background threads to be released in turn. Since the 'Thread xxx got signalled' message is outputted while the lock is held, and the condition variable ensures the lock is released for only one waiter at a time, the second 'Thread xxx got signalled' message will only be outputted after the first '... about to release the lock' one, despite the two second delay between that and the first thread being woken.

The fact waiting threads are released in an orderly fashion like this can make condition variables quite an attractive device to use. However, their big limitation is that the signalling thread has very little control over what threads it signals: in a nutshell, the choice is to signal either *one* waiting thread (the identity of which is unknown) or *all* waiting threads — in `TConditionVariableMutex`/`TConditionVariableCS` terms, this means calling either `Release` or `ReleaseAll`. Furthermore, if no thread is waiting at the time the condition variable is signalled, then the pulse is 'lost':

```
uses
  System.SysUtils, System.Classes, System.SyncObjs;

var
  Lock: TCriticalSection;
  CondVar: TConditionVariableCS;
begin
  Lock := TCriticalSection.Create;
  CondVar := TConditionVariableCS.Create;
  try
    CondVar.Release;
    TThread.CreateAnonymousThread(
      procedure
      begin
        Lock.Acquire;
        try
          case CondVar.WaitFor(Lock, 5000) of
            wrTimeout: WriteLn('Timed out!');
            wrSignaled: WriteLn('Got the signal');
          end;
        finally
          Lock.Release;
        end;
      end).Start;
    ReadLn;
  finally
    CondVar.Free;
    Lock.Free;
  end;
end.
```

In this example, the background thread's `WaitFor` call times out since `Release` was called by the main thread before `WaitFor` was called. This behaviour contrasts with events, with which a signal from a `SetEvent` call is never lost so long as the event object itself has not been destroyed:

```
var
  Event: TEvent; //or TLightweightEvent
begin
  Event := TEvent.Create;
  try
    Event.SetEvent;
    TThread.CreateAnonymousThread(
      procedure
      begin
        case Event.WaitFor(5000) of
          wrTimeout: WriteLn('Timed out!');
          wrSignaled: WriteLn('Got the signal');
        end;
      end).Start;
    Sleep(100);
  finally
    Event.Free;
  end;
end.
```

In this version, the signal will be taken.

Monitors (TMonitor)

A 'monitor' in a multithreading context is an object that combines a lock and condition variable. In Delphi, `TMonitor` is a record type defined in the `System` unit. We have already seen it in its role as a critical section mechanism that works against any class instance. For the condition variable part, it adds `Wait`, `Pulse` and `PulseAll` methods, which are equivalent to the `WaitFor`, `Release` and `ReleaseAll` methods of `TConditionVariableMutex` and `TConditionVariableCS`.

In practice, the wait/pulse functionality of `TMonitor` is not something you would use much (if at all) in application-level code. However, it can form the nuts and bolts of higher level primitives, and indeed, the `TCountdownEvent`, `TLightweightEvent` and `TLightwightSemaphore` classes we met earlier are all based around it. Beyond that, it can play a crucial part in solving 'producer/consumer' problems.

These arise when there are two or more threads in which one or more (the 'producers') creates data for the others (the 'consumers') to process. In the simplest cases, there are precisely two threads, one producer and one consumer; as the consumer waits on the producer to pass it something, so the producer will wait on the consumer to finish processing when it has new data to pass over.

If this scenario is hard to picture, imagine two people digging a hole in the ground, one doing the actual digging and the other taking away the earth in a wheelbarrow. In the first instance, the person with the wheelbarrow will be waiting on the digger to dig up some earth to take away; however, if the second person takes there time wheeling the earth away, then the digger will have to wait for her to return. Even if the digger wasn't worried about the wheelbarrower's whereabouts, there's only so much he can dig before surrounding himself with so much earth he can't dig any more!

To put this sort of situation into code, let's say there is just one piece of data at a time that is sent and waited upon. Wrapping the necessary code in a class, its interface could look like this:

```
type
  TThreadedData<T> = class
  strict private
    FData: T;
    FEmpty: Boolean;
    FLock: TObject;
  public
    constructor Create;
    destructor Destroy; override;
    function Read: T;
    procedure Write(const AData: T);
  end;
```

Here, the constructor will just initialise `FEmpty` to `True` and create the dummy object to lock against; the destructor will then just free the dummy object:

```
constructor TThreadedData<T>.Create;
begin
  inherited Create;
  FEmpty := True;
  FLock := TObject.Create;
end;

destructor TThreadedData<T>.Destroy;
begin
  FLock.Free;
  inherited Destroy;
end;
```

In the case of `Read`, we will wait on `FEmpty` becoming `False`, before changing it to `True` and issuing the signal; in the case of `Write`, we wait on `FEmpty` becoming `True`, before changing it to `False` and issuing the signal:

```
function TThreadedData<T>.Read: T;
begin
  TMonitor.Enter(FLock);
  try
    while FEmpty do
      TMonitor.Wait(FLock, INFINITE);
    Result := FData;
    FEmpty := True;
    TMonitor.PulseAll(FLock);
  finally
    TMonitor.Exit(FLock);
  end;
end;

procedure TThreadedData<T>.Write(const AData: T);
begin
  TMonitor.Enter(FLock);
  try
    while not FEmpty do
      TMonitor.Wait(FLock, INFINITE);
    FEmpty := False;
    FData := AData;
    TMonitor.PulseAll(FLock);
  finally
    TMonitor.Exit(FLock);
  end;
end;
```

The `Wait` call in both cases should be made in the context of a loop to cover the case of more than one waiting thread.

Putting this class to use, in the following example the main thread acts as the producer, feeding the consumer a list of strings:

```
uses
  System.SysUtils, System.StrUtils, System.Classes,
  System.SyncObjs;

procedure WorkerThreadProc(const AData: TThreadedData<string>);
var
  S: string;
begin
  { Enter a loop in which we wait for work, process the work given,
    wait for work, process the work given, and so on...}
  repeat
    S := AData.Read;
    if S = '' then Exit;
    Sleep(500); //simulate doing some 'proper' work...
    Write(ReverseString(S) + ' ');
  until False;
end;

var
  Data: TThreadedData<string>;
  FinishedSemaphore: TSemaphore;
  S: string;
begin
  Data := TThreadedData<string>.Create;
  FinishedSemaphore := TSemaphore.Create(nil, 0, 1, '');
  try
    //spin up the worker thread
    TThread.CreateAnonymousThread(
      procedure
      begin
        WorkerThreadProc(Data);
        FinishedSemaphore.Release;
      end).Start;
    //write the strings to process
    for S in TArray<string>.Create(
      'siht', 'si', 'a', 'omed', 'fo', 'rotinoMT') do
      Data.Write(S);
    //an empty string denotes the end of the process
    Data.Write('');
    //wait for the worker to finish
    FinishedSemaphore.WaitFor;
    WriteLn(SLineBreak + 'Finished!');
  finally
    FinishedSemaphore.Free;
    Data.Free;
  end;
end.
```

The output is `This is a demo of TMonitor`, each word coming up in turn after a short delay.

Futures

A ‘future’ is a value that is calculated ‘asynchronously’ in the background rather than ‘synchronously’ in the foreground while the calling thread waits on. At some later point, the calling code asks for the value; if the background calculation has completed, the result is returned, otherwise the calling thread is blocked until it can be given.

While Delphi does not provide a futures implementation in the box, it isn’t hard to write a simple one using anonymous methods. In the following example, a factory method wraps an anonymous function (the future delegate) in a threaded proxy (the future itself):

```
uses
  System.SysUtils, System.Classes;

type
  TFuture<T> = record
  strict private type
    TFutureImpl = class(TInterfacedObject, TFunc<T>)
    strict private
      FResult: T;
      FWorker: TThread;
    public
      constructor Create(const ADelegate: TFunc<T>);
      function Invoke: T;
    end;
  public
    class function Create(const ADelegate: TFunc<T>): TFunc<T>; static;
  end;

implementation

constructor TFuture<T>.TFutureImpl.Create(const ADelegate: TFunc<T>);
begin
  inherited Create;
  FWorker := TThread.CreateAnonymousThread(
    procedure
    begin
      FResult := ADelegate();
    end);
  FWorker.FreeOnTerminate := False;
  FWorker.Start; //have the delegate run in the background
end;

function TFuture<T>.TFutureImpl.Invoke: T;
begin
  if FWorker <> nil then
  begin
    FWorker.WaitFor;
    FreeAndNil(FWorker);
  end;
  Result := FResult;
end;

class function TFuture<T>.Create(const ADelegate: TFunc<T>): TFunc<T>;
begin
  Result := TFutureImpl.Create(ADelegate);
end;
```

This uses the ‘trick’ of implementing an anonymous method type directly — as we saw in chapter 4, an anonymous method is really an object accessed through an interface composed of a single method, `Invoke`. Therefore, the `TFunc<T>` method reference type, which is declared in `System.SysUtils` as

```
type
  TFunc<T> = reference to function : T;
```

is ‘really’ an interface that looks like this:

```
type
  TFunc<T> = interface
    function Invoke: T;
  end;
```

Our little futures implementation can be used like this:

```
uses IdHTTP;
```

```
function DownloaderForURL(const URL: string): TFunc<string>;
begin
  Result := function : string
            var
              IdHttp: TIdHttp;
            begin
              IdHttp := TIdHttp.Create(nil);
              try
                Result := IdHttp.Get(URL);
              finally
                IdHttp.Free;
              end;
            end;
end;

var
  Downloader: TFunc<string>;
  XML: string;
begin
  Downloader := TFuture<string>.Create(DownloaderForURL(
    'http://delphihaven.wordpress.com/feed/'));
  //...
  XML := Downloader(); //now retrieve the XML
```

In practice, this implementation is just a little too simple though: specifically, if lots of futures are created in quick succession, and each delegate takes only a short period of time to complete, then the overhead of creating a new thread for each one will cause the code to take *far longer* to complete than if the operations were performed one after the other, in a single threaded fashion. We need, then, to be a bit cleverer about how we allocate new threads.

The general solution to this problem is to use a 'thread pool'. In this, instead of each future delegate being passed to a worker thread directly, it is put in a queue; worker threads are then written to follow a looping pattern, waiting on a task being put on the work queue before picking it off, performing it, waiting on another task and so on. To avoid the problem of too many threads being spun up, their number is capped, either by a fixed amount (e.g. twice the number of processor cores) or by some algorithm that makes an educated guess given the nature of the queued tasks so far. If there are more tasks than worker threads at any one time, the overspill is then made to wait in an orderly fashion.

Thread pool considerations

As with futures themselves, the Delphi RTL does not provide a stock thread pool implementation. Nonetheless, you have all the tools you need to make one yourself. Specifically, the parts needed are a queue, a signalling method to let worker threads know both when a task needs performing and when the queue has been shut down, and a lock to ensure tasks are added and taken off the queue in a thread-safe manner. There are however a few complications to consider —

Exceptions: in the one thread per future model, it isn't crucial to explicitly handle an exception raised inside the delegate function, since while an exception would kill the thread, that doesn't matter since the thread would die with the future anyhow. Once a thread pool is used though, one thread will serve many futures; an unhandled exception in a delegate will therefore have the effect of killing off the thread for any subsequent tasks too. If unhandled exceptions are then raised in each of the remaining worker threads too, you will be left with no worker threads at all!

Cancelling a task explicitly: if future delegates are queued, then it makes sense to allow taking a task off of the queue if its return value is no longer of interest. Otherwise, it will stay in line and so block the execution of tasks subsequently added to the queue that *are* still of interest.

Cancelling a task implicitly: a subtle variant of the previous point is when the future is allowed to go out of scope without being invoked:

```
var
  SomeCalc: TFunc<string>;
begin
  SomeCalc: TFuture<string>.Create(
    function : string
    begin
      //do something...
    end);
  //do something else without invoking SomeCalc...
end;
```

In such a case, if the future delegate is still on the task pool, it makes sense for the futures implementation to cancel it.

The number of worker threads: the question arises as to how many threads the task pool should start up. Bluntly put, there is no generally correct answer to this question; while limiting the pool to fewer threads than there are CPU cores would be a pointless economy, the number beyond that depends upon the nature of the tasks being performed.

For example, when there are a lot of tasks that (say) download things from the internet, spinning up a relatively large number of threads will be fine, since the hard work is performed not on the local computer, where CPU resources are scarce, but external servers. In such a case, worker threads will spend most their time just blocking (quietly waiting). In contrast, if the majority of tasks are complicated calculations, then creating more threads than CPU cores is likely to only harm performance, especially if each calculation is performed independently from the others.

Because of this, a stock thread pool implementation will usually try and be clever, using heuristics to progressively increase the number of workers to a certain level that is itself dynamically determined. When writing your own thread pool, there's usually no need to be particularly clever though, since you know the sorts of tasks that will be sent to it, and so, should be able to empirically determine a reasonable thread allocation strategy. For the futures case, we will just take the simple option of fixing the number of worker threads to double the number of CPU cores; as a further simplification, they will all be spun up at the outset.

Implementing the thread pool

Since every future will share the same thread pool, we will implement the pool as a 'singleton' (since instance object). As one of the quickest ways to implement a singleton in Delphi is to define a record type with static members, we'll do just that:

```
uses
  System.SysUtils, System.Classes, System.SyncObjs;

type
  TFutureTaskToken = type Pointer;

  TFutureTasks = record
  strict private type
    PNode = ^TNode;
    TNode = record
      NextNode: PNode;
      Task: TProc;
    end;
  strict private
    class var FHeadNode, FTailNode: PNode;
    class var FTerminated: Boolean;
    class var FThreadsLeft: TCountdownEvent;
    class var FQueueLock: TObject;
    class constructor Create;
    class destructor Destroy;
    class function WaitFor(out ATask: TProc): Boolean; static;
  public
    class function Add(const ATask: TProc): TFutureTaskToken; static;
    class function Cancel(ACancellationToken:
      TFutureTaskToken): Boolean; static;
    class procedure CancelAll; static;
    class procedure Terminate; static;
  end;
```

The fact we want to allow cancelling a task leads to using a linked list rather than `TQueue`, since the latter does not allow dequeuing any item but the first one in line. For the actual cancelling, the `TFutureTaskToken` returned by `Add` is an 'opaque pointer' type in the sense calling code does not have to know what it 'really' points to — rather, it's just a token that can be passed to `Cancel` to get the corresponding task off the queue.

For simplicity, all worker threads will be created at the off:

```
class constructor TFutureTasks.Create;
var
  I: Integer;
begin
  FQueueLock := TObject.Create;
  FThreadsLeft := TCountdownEvent.Create(1);
  for I := 1 to CPUCount * 2 do
    TThread.CreateAnonymousThread(
      procedure
      var
        Task: TProc;
      begin
        FThreadsLeft.AddCount;
        try
```

```
          while WaitFor(Task) do
            Task;
        finally
          FThreadsLeft.Signal;
        end;
      end).Start;
end;
```

The thread procedure first increments the active threads counter before entering into the 'wait for task'/'perform task' loop. Our private `WaitFor` method returning `False` will mean the queue has shut down, at which point the thread decrements the active thread counter and terminates.

`WaitFor` itself is implemented like this:

```
class function TFutureTasks.WaitFor(out ATask: TProc): Boolean;
var
  OldNode: PNode;
begin
  TMonitor.Enter(FQueueLock);
  try
    while FHeadNode = nil do
    begin
      TMonitor.Wait(FQueueLock, INFINITE);
      if FTerminated then Exit(False);
    end;
    OldNode := FHeadNode;
    FHeadNode := OldNode.NextNode;
  finally
    TMonitor.Exit(FQueueLock);
  end;
  ATask := OldNode.Task;
  Dispose(OldNode);
  Result := True;
end;
```

This just implements the pattern we met when looking at `TMonitor` earlier, only with an added check for the `FTerminated` field becoming `True`. This will be case on Terminated being called:

```
class procedure TFutureTasks.Terminate;
begin
  FTerminated := True;
  TMonitor.PulseAll(FQueueLock);
  CancelAll;
end;
```

We need to use `PulseAll` here given more than one worker thread may be waiting on the monitor, and we want them all to now recheck the value of `FTerminated`. `CancelAll` (which can be called separately) will remove all queued items from the queue without actually performing them:

```
class procedure TFutureTasks.CancelAll;
var
  Next: PNode;
begin
  TMonitor.Enter(FQueueLock);
  try
    while FHeadNode <> nil do
    begin
      Next := FHeadNode.NextNode;
      Dispose(FHeadNode);
      FHeadNode := Next;
    end;
    FTailNode := nil;
  finally
    TMonitor.Exit(FQueueLock);
  end;
end;
```

`Terminate` itself will be called on closedown, and so, in the record type's destructor. Depending on the tasks, in may be OK just to let the operating system terminate everything, i.e. not explicitly finalise anything ourselves. However, as there could in principle be database connections or the like that need to be deliberately closed, we will terminate gracefully, though with a limit to how long we will wait:

```
class destructor TFutureTasks.Destroy;
begin
  Terminate;
  FThreadsLeft.Signal;
  if FThreadsLeft.WaitFor(5000) <> wrSignaled then Exit;
  FQueueLock.Free;
  FThreadsLeft.Free;
end;
```

This just leaves the `Add` and `Cancel` methods. The former will initialise a new node, get a lock on the queue by entering the monitor, add the node, then pulse to one waiting worker thread. Finally, an opaque pointer to the node will be returned as the cancellation token:

```
class function TFutureTasks.Add(const ATask: TProc): TFutureTaskToken;
var
  Node: PNode;
begin
  Node := AllocMem(SizeOf(TNode));
  try
    Node.Task := ATask;
    TMonitor.Enter(FQueueLock);
    try
      if FTerminated then
        raise EInvalidOperation.Create(
          'Cannot add to a terminated pool');
      if FHeadNode = nil then
        FHeadNode := Node
      else
        FTailNode.NextNode := Node;
      FTailNode := Node;
      TMonitor.Pulse(FQueueLock);
    finally
      TMonitor.Exit(FQueueLock);
    end;
  except
    Dispose(Node);
    raise;
  end;
  Result := TFutureTaskToken(Node);
end;
```

Correspondingly, `Cancel` is implemented by taking a lock on the queue, before scanning for the node. Once found, it is taken off the queue, the linked list repaired, the lock relinquished, and `True` returned. If the node wasn't found, then the lock is relinquished and `False` returned:

```
class function TFutureTasks.Cancel(
  ACancellationToken: TFutureTaskToken): Boolean;
var
  Previous, Current: PNode;
begin
  Previous := nil; //keep compiler happy
  TMonitor.Enter(FQueueLock);
  try
    Current := FHeadNode;
    while Current <> nil do
      if TFutureTaskToken(Current) = ACancellationToken then
      begin
        if Current = FHeadNode then
          FHeadNode := Current.NextNode
        else
          Previous.NextNode := Current.NextNode;
        Dispose(Current);
        Exit(True);
      end
      else
      begin
        Previous := Current;
        Current := Current.NextNode;
      end;
  finally
    TMonitor.Exit(FQueueLock);
  end;
  Result := False;
end;
```

As such, the task pool could be used independently of futures, albeit with care taken to ensure no exception will escape the task procedure:

```
TFutureTasks.Add(
  procedure
  begin
```

```
    try
      //do something...
    except
      //handle exception intelligently...
    end;
  end);
```

Let’s now look at re-implementing the future class itself.

A better future

As before, the future will be an anonymous method implemented by an explicit class:

```
TFuture<T> = record
strict private type
  TFutureImpl = class(TInterfacedObject, TFunc<T>)
  strict private
    FCompletedEvent: TLightweightEvent;
    FFatalException: TObject;
    FInvokeTimeout: LongWord;
    FResult: T;
  public
    constructor Create(const ADelegate: TFunc<T>;
      out ACancellationToken: TFutureTaskToken;
      AInvokeTimeout: LongWord);
    destructor Destroy; override;
    function Invoke: T;
  end;
public
  class function Create(const ADelegate: TFunc<T>;
    AInvokeTimeout: LongWord = INFINITE): TFunc<T>; overload; static;
  class function Create(const ADelegate: TFunc<T>;
    out ACancellationToken: TFutureTaskToken;
    AInvokeTimeout: LongWord = INFINITE): TFunc<T>; overload; static;
end;
```

The factory methods just create the ‘method’:

```
class function TFuture<T>.Create(const ADelegate: TFunc<T>;
  AInvokeTimeout: LongWord): TFunc<T>;
var
  AToken: TFutureTaskToken;
begin
  Result := TFutureImpl.Create(ADelegate, AToken, AInvokeTimeout);
end;

class function TFuture<T>.Create(const ADelegate: TFunc<T>;
  out ACancellationToken: TFutureTaskToken;
  AInvokeTimeout: LongWord): TFunc<T>;
begin
  Result := TFutureImpl.Create(ADelegate, ACancellationToken,
    AInvokeTimeout);
end;
```

For `TFutureImpl` itself, we have two new fields compared to the previous version, namely `FFatalException` and `FCompletedEvent`. The first will be for the task procedure to store any exception raised in the delegate. Once the user requests the future’s return value, the exception (or a copy of it, given the future could be invoked multiple times) will be re-raised:

```
function TFuture<T>.TFutureImpl.Invoke: T;
var
  ExceptObject: TObject;
begin
  if FCompletedEvent.WaitFor(FInvokeTimeout) = wrTimeout then
    raise ESyncObjectException.Create('Future timed out');
  if FFatalException = nil then Exit(FResult);
  ExceptObject := FFatalException;
  //copy the exception in case we are called again
  if ExceptObject is Exception then
    FFatalException := ExceptClass(ExceptObject.ClassType).Create(
      Exception(ExceptObject).Message)
  else
    FFatalException := ExceptObject.ClassType.Create;
  raise ExceptObject;
end;
```

`FCompletedEvent` will serve a dual purpose. In the normal case, `Invoke` will wait on it being signalled so that it can return the delegate result. In the case of the future being implicitly cancelled however, `Invoke` will never be called. There, the *task procedure* will instead wait on `FCompletedEvent` being signalled by `TFutureImpl.Destroy`:

```
destructor TFuture<T>.TFutureImpl.Destroy;
var
  DoFree: Boolean;
begin
  TMonitor.Enter(FCompletedEvent);
  try
    DoFree := FCompletedEvent.IsSet;
    if not DoFree then FCompletedEvent.SetEvent;
  finally
    TMonitor.Exit(FCompletedEvent);
  end;
  if DoFree then FCompletedEvent.Free;
  FFatalException.Free;
  inherited;
end;
```

A lock must be used because while an event's `SetEvent` method is thread-safe (it would be pretty useless otherwise!), its destruction is *not*.

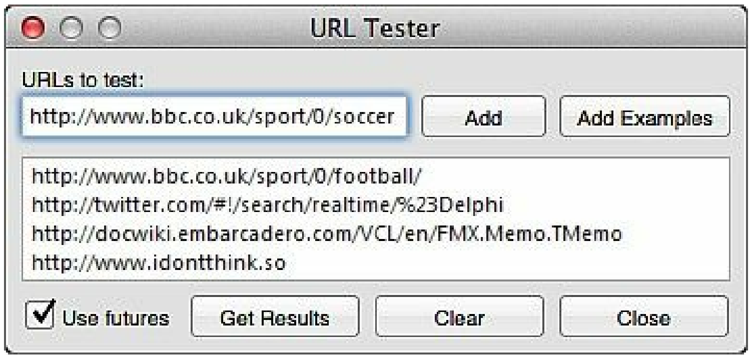
Similar to how the thread procedure in the naïve implementation was defined inline inside the future's constructor, so the task procedure is defined similarly:

```
constructor TFuture<T>.TFutureImpl.Create(
  const ADelegate: TFunc<T>;
  out ACancellationToken: TFutureTaskToken;
  AInvokeTimeout: LongWord);
begin
  inherited Create;
  FCompletedEvent := TLightweightEvent.Create;
  FInvokeTimeout := AInvokeTimeout;
  ACancellationToken := TFutureTasks.Add(
    procedure
    var
      DoFree: Boolean;
    begin
      TMonitor.Enter(FCompletedEvent);
      try
        DoFree := FCompletedEvent.IsSet;
        if not DoFree then
        begin
          try
            FResult := ADelegate;
          except
            FFatalException := AcquireExceptionObject;
          end;
          FCompletedEvent.SetEvent;
        end;
      finally
        TMonitor.Exit(FCompletedEvent);
      end;
      if DoFree then FCompletedEvent.Free;
    end)
end;
```

Use of the thread pool version of `TFuture<T>` is exactly the same as the naïve version, only with the ability to request cancellation if a future hasn't already been taken off the task pool:

```
var
  Future: TFunc<string>;
  Token: TFutureTaskToken;
begin
  Future := TFuture<string>.Create(SomeFunc, Token);
  //...
  TFutureTasks.Cancel(Token);
```

You can find the complete code for this futures implementation in the book's source code repository. With it comes a demo that uses it for a simple URL tester application:



http://www.bbc.co.uk/sport/0/soccer

http://www.bbc.co.uk/sport/0/football/
http://twitter.com/#!/search/realtime/%23Delphi
http://docwiki.embarcadero.com/VCL/en/FMX.Memo.TMemo
http://www.idontthink.so

When the 'Get Results' button is clicked, a dialog box reports whether each URL added to the list box is valid, is subject to redirection, or is invalid. As URLs are added to the list box in the first place, tester functions are added to the futures task pool. By the time the user comes to click 'Get Results', therefore, there is a good chance that at least some of URLs have already been tested for, making the dialog box show up immediately. Whether or not all testing functions have finished executing is completely transparent to the user interface code however — for all the UI code knows, it just has a series of functions to execute at the time the 'Get Results' button is clicked.